

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS no. 1371

HOLOPORTATION OF HUMAN MODELS

Oleg Jakovljević

Zagreb, June 2017.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1371

HOLOPORTACIJA MODELA LJUDI

Oleg Jakovljević

Zagreb, Lipanj 2017.

Zagreb, 3 March 2017

MASTER THESIS ASSIGNMENT No. 1371

Student: **Oleg Jakovljević (0036475226)**
Study: Computing
Profile: Computer Science

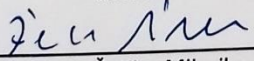
Title: **Holoportation of Human Models**

Description:

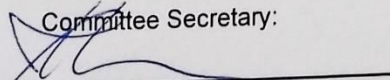
Investigate depth acquisition and motion sensing input devices like Microsoft Kinect and accompanying file formats for data transfer. Investigate possibilities of data transfer for the obtained three-dimensional scanned models, especially for human models. Design and implement holoportation of human characters to distant locations. Implement a framework for analysis and comparison of the investigated models. Show examples of results. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use the game engine Unity. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Issue date: 10 March 2017
Submission date: 29 June 2017


Mentor:


Full Professor Željka Mihajlović, PhD

Committee Secretary:


Assistant Professor Tomislav Hrkać, PhD

Committee Chair:


Full Professor Siniša Srbljić, PhD

Zagreb, 3. ožujka 2017.

DIPLOMSKI ZADATAK br. 1371

Pristupnik: **Oleg Jakovljević (0036475226)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Holoportacija modela ljudi**

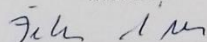
Opis zadatka:

Proučiti uređaj Kinect za trodimenzionalno uzorkovanje prostora te podatke koje generira pri snimanju. Proučiti razne mogućnosti načina prijenosa podataka dobivenih 3D senzorom a posebice obratiti pažnju na uzorkovanje modela ljudi. Razraditi i implementirati holoportaciju ljudskih likova na udaljenu lokaciju. Načiniti programsku implementaciju i razviti pripadnu arhitekturu programskog rješenja. Na različitim primjerima prikazati ostvarene rezultate. Načiniti ocjenu rezultata i implementiranih algoritama.

Izraditi odgovarajući programski proizvod. Koristiti grafički programski pogon Unity. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 10. ožujka 2017.
Rok za predaju rada: 29. lipnja 2017.

Mentor:



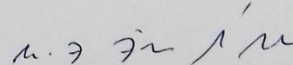
Prof. dr. sc. Željka Mihajlović

Djelovođa:



Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:



Prof. dr. sc. Siniša Srblijić

Contents

1. Introduction	1
2. Background	2
2.1 Project task	3
2.2 System hardware	3
3. System development	7
3.1 Researched methods	7
3.2 Developed components	14
3.2.1 Networking	16
3.2.2 Depth data processing	17
3.2.3 Color data processing	20
3.2.4 Other components	22
4. Developed system	24
4.1 Video and audio data sender	25
4.1.1 Sender components	25
4.2 Data receiver and visualizer	33
5. Conclusion	38
Abstract	40
Bibliography	41

1.Introduction

This thesis is based around a project aimed to enable real time holoportation of human models. The term holoportation defines 3D holograms, displayed in mixed reality, used for real time communication between remote users. Implementation of the system was based on usage of certain hardware components, namely the Microsoft Kinect v2 depth camera and the Microsoft HoloLens mixed reality glasses. During the development the project was divided into sections, as each section required various methods to be researched before the optimal solution could be found. The thesis describes the technological problem set by the task at hand, the solution that was determined to be the best, as well as the other considered options. Each phase of the project required research and implementation, and due to many unknown variables the development process was not concretely defined. The result is a proof of concept for a holographic communication platform, uniting a number of solutions and components which are easily expandable to a commercially usable product.

2. Background

Many difficulties and complications arise when discussing real time communication between two remote components both of which have many responsibilities. The set requirements state one-way communication between a desktop machine with a depth sensor and portable smartglasses which would be used to display a user recorded by the depth sensor. It was estimated that at the current phase, one way communication would be easier to implement and sufficient to demonstrate the feasibility of the project and the efficiency of technological solutions used. Due to the nature of the task and the relative uncertainty of the implementation methods, it was not possible to enforce a rigid plan of development. There are several points of interest when it comes to dissecting the problem. The first and most obvious issue is the limited network communication bandwidth. In order to make the system usable in real world cases, the bandwidth requirement has to be tailored to meet real world expectations. Secondly, an important factor to be considered are the processing power of the server and client devices, namely the desktop computer used to record and the mobile smartglasses used to display the holoported user. These requirements lead to another issue to be considered: the latency of the system. In order to perceive a natural conversation, the client (using the smartglasses) has to receive real-time data with low latency, both for visual and audio data. Considering these requirements makes it possible to determine which methods are appropriate and which would cause these constraints to be violated. Since most of the ideas considered are quite novel and not tested in a use case appropriate for this system, many components had to be implemented so that they could be evaluated and potentially used in the final solution. The following chapters cover the initial project task and describe the hardware used in the implementation.

2.1 Project task

Creating a system for real time holoportation of human models is a problem with many unknown variables, so a modular approach was selected to subdivide the components in development and enable independent testing of components. A project was designed, specifying that two programs should be developed: a data sender application run by a desktop computer with a depth camera recording a person, and a data receiver run by a mobile headset displaying a virtual object of the user in real space (mixed reality). This system must work real time and the latencies involved should not hinder the communication properties of the platform (i.e. the delay must not be noticeable when communicating in real time). The Unity3D engine was chosen as the base for the system, due to its versatility and a wide range of supported hardware and software. Unity3D also enables a dynamic environment for testing and agile development.

Building the system that performs desirably requires finding the best components that fit the requirements. It was initially determined that those key components will be data capture, mesh construction, data compression, networking and data reconstruction on the client side. Some of those components were changed as development progressed and deviated from the original course. It was also anticipated that some third-party software would be used, in the form of dynamic libraries, static libraries or precompiled executables.

2.2 System hardware

The basic hardware of the system, in its one-way configuration, consists of the Microsoft Kinect V2 depth camera on the server side (data provider) and the Microsoft HoloLens smartglasses on the client side (data receiver). These devices were primarily selected because they are supported by the Unity3D engine development environment which allows easy integration and testing. Kinect v2 is supported by a Unity3D plugin since July 2014 and HoloLens support came with Unity3D 5.5.0 released on Nov 30, 2016 [1]. Kinect v2 is supported by an adapter plugin which makes the native

Kinect code available in Unity3D, for scripting in C#. This adapter is well built and thus little performance is lost when using it, which had proven to be crucial during the later stages of development, when performance became one of the primary issues in the system. Some parts of the plugin had to be modified to better fit the needed use case in the system.

Kinect v2 SDK support included in Unity3D offers access to raw data streams for the full HD resolution RGB camera, the depth image (resolution 512x424 pixels) and the audio streams. It also provides processed data, such as skeleton tracking (with 25 joints per person) with position and orientation data, and allows this data to be used in Unity3D easily. Using the skeleton tracking and person recognition, it's possible to perform efficient background removal that is more accurate than background removal done using RGB imaging alone. The API examples provided starting points for various components that were developed during the different phases of the project. There are several advantages to using a depth camera in contrast to one or more RGB cameras. Depth cameras enable simple mesh reconstruction and accurate background removal, which is required in order to reduce the amount of network traffic and render the user in augmented reality with a degree of precision. Furthermore, using multiple depth cameras to obtain a full 3D real time scan of the user is a possible upgrade of the system. Figure 2.1 shows the Microsoft Kinect V2 depth sensor and its main components.

Kinect sensors use a space mapping technique called *structured light* which relies on an infrared matrix of predefined dimensions projected by a light source in the space. Once the infrared stereo cameras records an image, the infrared points are located and compared to the orthographically projected matrix, which tells how much each depth point is displaced from its neutral position. This is how the depth camera can define the distance of each point to the sensor, which produces the depth frame. The sampled data is fed into a classifier which determines where the person is located, relative to the depth camera. The person's body position is stored as joint positions (for arms, legs, head and other joints) and their orientations, so that the body tracking does not depend on the person's height and position.

Kinect 2 - Specs

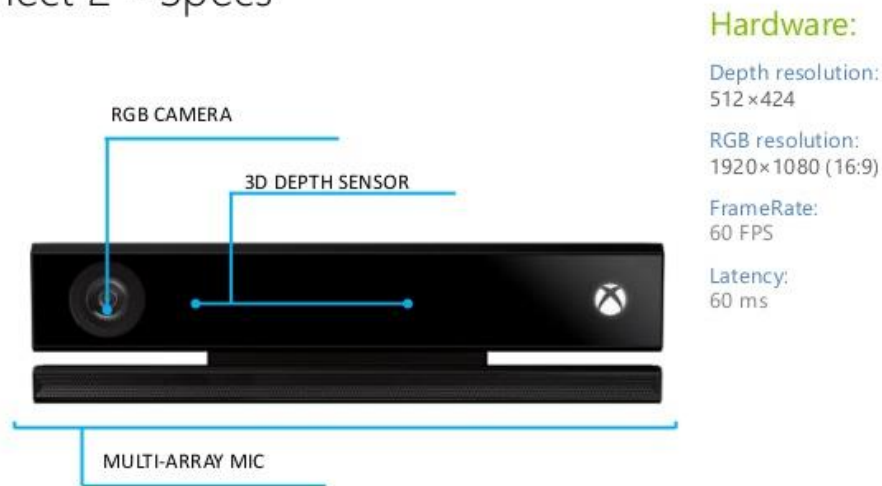


Figure 2.1 – Kinect v2 description

The Microsoft HoloLens holographic device features multiple sensors and processing units, such as a depth camera, IMU (inertial measurement unit), HD video camera, microphones and ambient light sensors. Unlike most previous devices, it is completely independent of any other devices and does not have to be in any way tethered or connected to a desktop computer. It uses onboard processing units, more specifically a Holographic Processing Unit and an Intel 32 bit architecture compatible processor [2]. It has 2 gigabytes of RAM memory and 64GB of flash memory for the OS and various applications. The device is also compatible with Unity3D which enables simple development of mixed reality applications. However, due to the fact that its OS is based on the Universal Windows Platform, there are restrictions in libraries and dependencies which can be used. For example, all Windows compiled libraries need to be recompiled for use with the UWP, and some libraries used in Unity3D are not available either. Although development was significantly streamlined and simplified by HoloLens' compatibility with Unity3D, these restrictions also had to be taken into account. Figure 2.2 shows a mixed reality headset and its principle components. The HoloLens device also has multiple sensors which help it display the user in mixed reality and give its user the feeling of presence and realism. Among its sensors it has a depth camera which is used to map the user's surroundings (usually the room that he is in) so that virtual objects could be placed in real

space and displayed relatively to the floor or some other part of the room, and even allows the virtual objects to be (partially) occluded by objects such as tables or chairs. The inertial measurement unit works together with other sensors to make sure that the virtual objects are seamlessly blended with the real mapped surroundings even in quick user or head movements. A holographic headset such as the HoloLens most commonly uses a transparent display to combine projected images and color with light from the background in order to produce an overlay of the virtual item on the background.

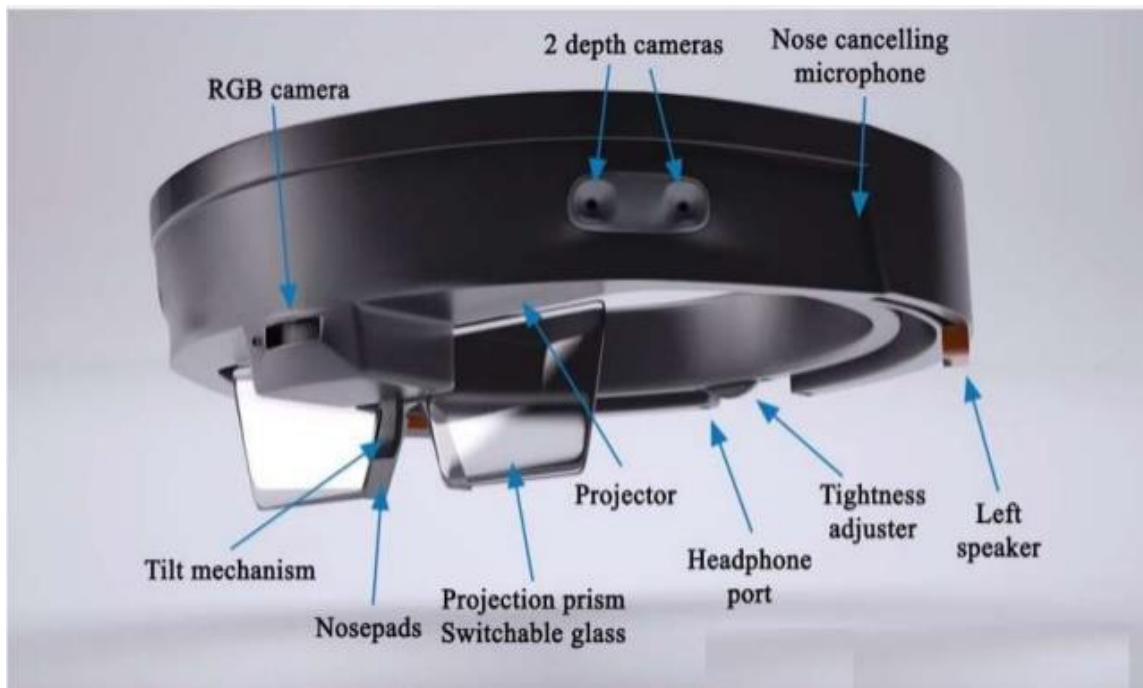


Figure 2.2 – Mixed reality headset components

The selected components are readily available; therefore the system was designed to use them initially. It was, however, planned to develop the system as a collection of abstract components which can be readily replaced as new hardware needs to be supported. However due to the nature of this technology, the field of view is limited to less than 40 degrees, which limits user immersion.

3. System development

Unity3D had been chosen as the platform for the project due to its versatility and compatibility with available hardware. Unity3D is a game engine with a game development platform built around it, allowing simple development of games and graphical applications and their deployment to many desktop and mobile platforms. It allows implementation of complex functionalities with relatively little additional work, because the existing engine offers most of the base functionality. Many solutions already exist as plugins or scripts for Unity3D because of its large user base and many useful components can be used in a project without much adaptation. There are similar engines and game development frameworks such as Unreal Engine, however, existing plugins and support for the components used in this system made Unity3D the more suitable choice. One of the main reasons for using the Unity engine in this project was, however, the familiarity of the development team with the platform and our experience with working with Unity.

3.1 Researched methods

The following chapter gives a brief overview of the options that were considered during the development, to explain how the final product was developed and which decisions lead to the way components are implemented in the final version. It was originally planned to use multiple native plugins for the system as it was expected that these would offer the needed interoperability with other existing components and better performance. A wireframe application was built in Unity3D, with calls to native plugin functions in dynamic link libraries written in C++, and several plugin projects were prepared for use as native plugins (dynamic libraries). Figure 3.1 shows the planned system operation flow with multiple components integrated.

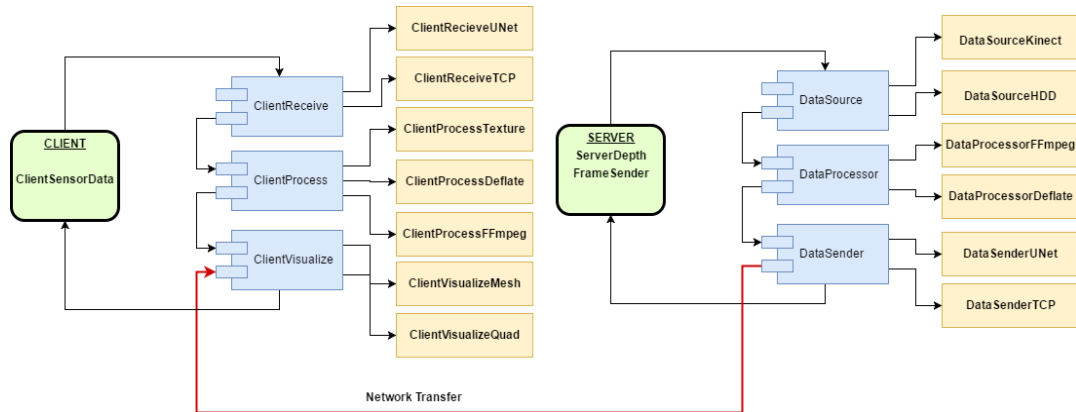


Figure 3.1 – Planned system layout

The planned plugins were Background Removal, Compression and Networking, and these components were meant to be called sequentially, pipelining data from one to the next one. Although some of these were used during various research and development phases, none of them made it to the final system. Let's have a look at the problems encountered.

Data representation

When it comes to **depth data**, the first question is how to store and transfer the data. One of the most popular methods of representing depth sensor data is using **point clouds**, where each of the cloud elements represents a point in the 3D space. This piece of space is defined by its origin point and its bounds (if there are any). In the case of the Kinect V2 depth sensor, its precision allows for 11 bits of depth data per point, which is interpreted as the distance from the sensor. This limits the precision and maximum distance from the sensor. Its range for high quality tracking is officially rated [1] at 0.5 to 4.5 meters, however the sensor still works at longer distances, although sacrificing accuracy and reliability [3]. One of the first ideas was using third party libraries for point cloud compression, such as the Point Cloud Library [4]. This had proven to be inefficient due to the complexity of those libraries and the low efficiency of the compression. The

image below shows the results produced by point cloud visualization with each point coloured with its respective color texture pixel.

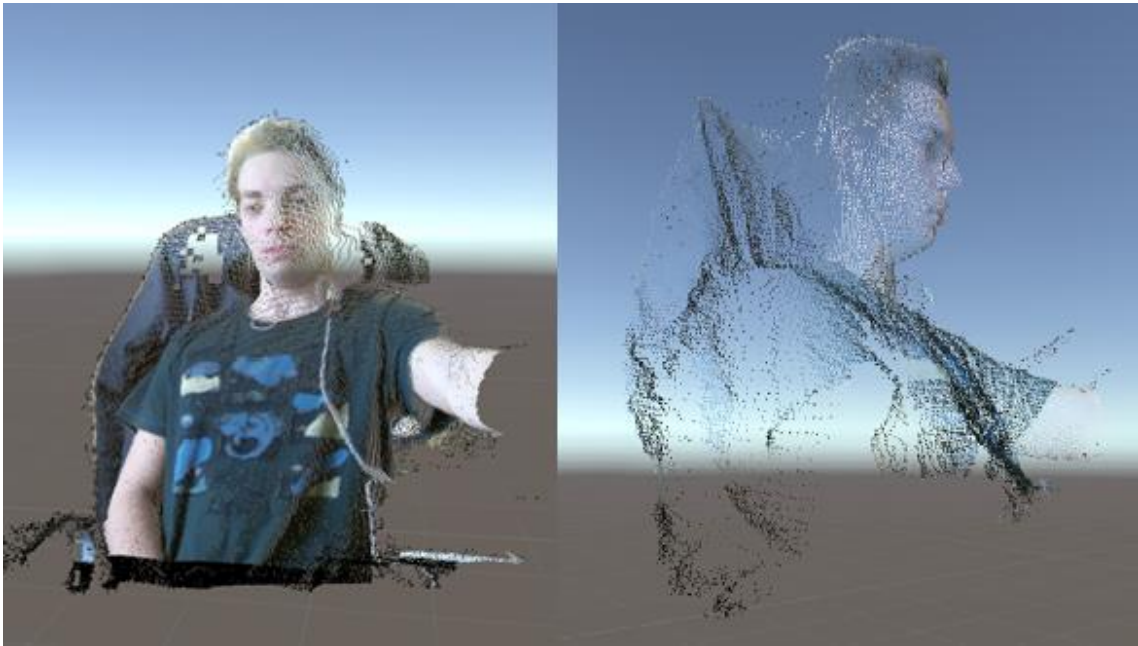


Figure 3.2 – Point cloud visualization

Though point clouds are an intuitive method of depth data representation, they have certain considerable disadvantages. Firstly, this is the “raw” representation of the data captured by the sensor, unprocessed and unoptimized. This means that if data represented by point clouds are sent over the network, all the processing has to be done by the receiver (assuming that the receiver is tasked with displaying the data), in addition to the potentially bigger data packet which has to travel over the network.

Another approach to data representation is using a **mesh** reconstructed from the depth data. This delegates processing to the sender (assuming, again, that the sender handles data recording and the receiver handles data displaying, in our case) and allows the data to be stored in a Unity3D *Mesh*, which can easily be applied to objects in scenes. Unity’s *Mesh* objects contain data such as vertex arrays, normals, UV texture maps, triangles and even tangents. This format is used to display general 3D objects in scenes. An additional advantage of this method is the possibility of applying background removal, which reduces the amount of data that needs to be sent over the

network. The quality and speed of mesh reconstruction from depth data varies greatly on the algorithm used. For this system, the method used for mesh creation is the simple triangle mesh creation found in the Unity3D Kinect v2 SDK Samples which can be obtained from the Unity Asset Store.

The third method of data representation is using a conventional **image** to represent the depth data (most intuitively represented by a grayscale image). Many algorithms exist for handling images, streaming them in real time and compressing them. This approach favors network transfer optimization and combines some of the perks of the other two methods listed above. *Image 1* shows the depth and color frames encoded into bitmaps with the depth sensor resolution of 512x424 pixels. Using bitmaps for depth images it is possible to perform background removal on the server side, as soon as the image is captured, and send a compressed video stream over the network, further reducing the amount of sent data. Image 3.3 shows the depth frame represented by a grayscale image, with darker tones meaning that the point is sampled close to the sensor and lighter tones mean that the point is sampled at a longer distance from the depth camera.

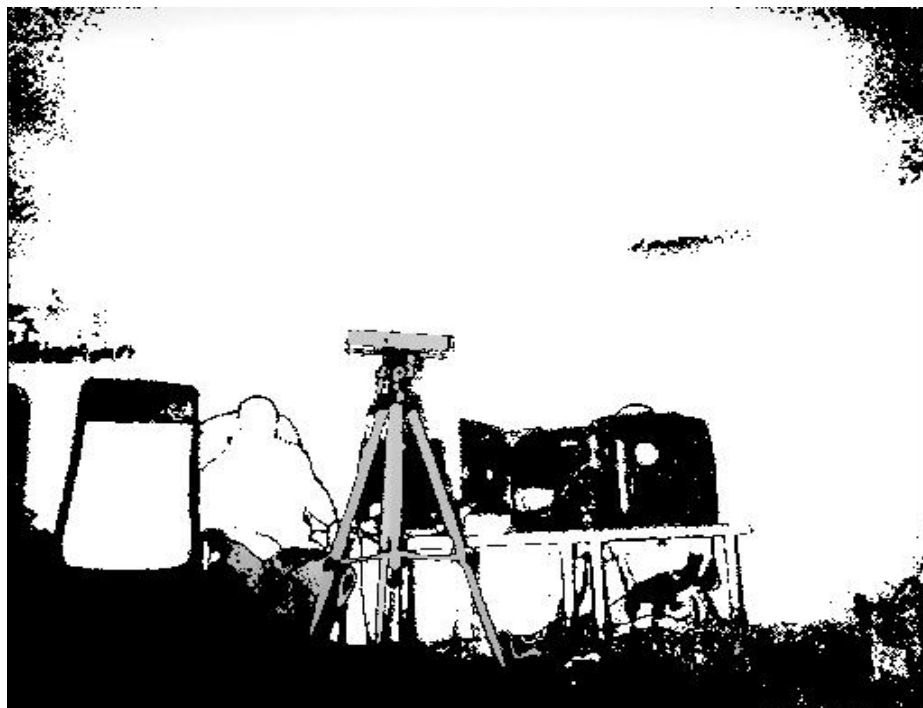


Figure 3.3 – Depth frame shown as a grayscale image

For the color texture, both the full HD (1920x1080 pixels) resolution and the depth image resolution (512x424 pixels) color texture are best stored in a *Texture2D*, Unity3D's texture format. This container format provides many useful functions for handling image textures and applying them to meshes. Some of these functions include: texture compression, serialization (to a byte array), JPG and PNG format encoding, a pointer to the raw texture data, a method to obtain or change the color of an individual pixel and many other.

Data Processing

Once the initial idea with using an external point cloud management library was abandoned due to complexity and low modularity, one of the ideas was to abandon real-time data transfer entirely and perform a *pre-scan* of the user to create a (rough) mesh which can be transferred once to the receiver. This, however, is less intuitive and brings more problems into the equation. More concretely, this means the recorded user is shown to the receiver as an avatar, which is difficult to animate if the animations should look natural. One of the most important unanswered questions here is how to actually rig and animate the user's avatar, because no reliable methods for real time rigging have been developed yet. Facial expressions are also almost impossible to mimic with avatars, which is why a plain mesh is much easier to work with and has a much more natural feel.

There are several operations that need to be performed on the data before it can be displayed on the receiver mixed reality headset. In order to keep the network traffic down, the data needs to be compressed, and the user needs to be segregated from the background in the recording, to place him in the viewer's environment in a realistic manner. For the **background removal**, the script *UserMeshVisualizer* from the "Kinect v2 Examples with MS-SDK" Unity package was utilized. This script obtains the depth data and color data, creates the mesh and displays the mesh with the texture applied in the scene. Figure 3.4 shows the scene view of the main sender scene with the user's mesh visualized. The script showed performance issues so optimizations were added, these are discussed in section 4 of this thesis.

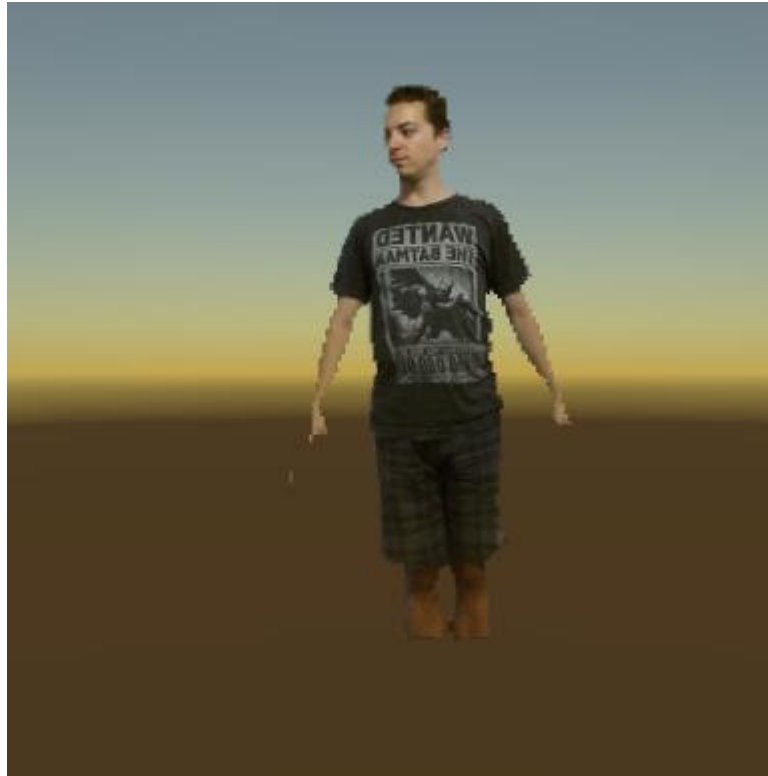


Figure 3.4 – Scene view of the sender-side visualization

When discussing **compression** methods, an efficient way of compressing both mesh or depth data and color textures is needed. For the depth data, many generic compression algorithms were considered, all of them lossless however. Among them it's worth to point out run-length encoding (which would be very inefficient with mesh data which is stored in a standard way (because x, y, and z coordinates often differ), LZ78 and Deflate. The compression algorithms are required to run quickly in real time and reduce the size of the data significantly to meet the requirements of the system. Color data compression methods include JPG and PNG format encoding, methods for encoding Texture2D objects to these formats are included in Unity3D's API.

Another course of development was to use an external program to form an encoded video stream which could be streamed efficiently. This works in conjunction with depth data represented as bitmaps, because thus both the color and depth data can be streamed as two (or even one) video stream.

The considered program in question is FFmpeg, a free framework that provides media processing and handling libraries [5]. FFmpeg offers many solutions for working with audio and video data, such as codecs, multiplexing and demultiplexing, transcoding and many options such as hardware acceleration. The project is distributed in several variants: the complete OS-dependent executable build, a dynamic link library build and a development project containing the source code. Arguably the most popular of these is the executable build, due to the fact that it is the best documented form of the framework and that it has a relatively large user base. The FFmpeg framework is suitable because it offers a “black box” solution [6] for encoding and decoding video data, as well as streaming it over the network. It also supports hardware acceleration methods such as Intel Quick Sync Video and Nvidia NVEnc, which are available for most of the frequently used CPUs and GPUs. Although FFmpeg was ultimately not used in the system, some components and solutions were developed during the research phase, and these are further described in chapter 3.2. Data delivery became the next focus of the project.

Soon, two general approaches to **data delivery** were considered: network transfer of point clouds and network depth image stream. The second method offered an advantage because the depth image stream can be represented as a 512x424 grayscale video (8 bits per pixel) along with the color stream which can use the same resolution and 32 bits per pixel. This allows the usage of well-established video streaming protocols and codecs, such as the mpeg-4 codec which supports delta compression. The downside of this method is that only one depth camera can be used per stream, so the system is not easily upgradable to use multiple depth cameras. Also, this configuration requires the usage of at least two video streams, unless depth data is packed into the alpha channel of the RGB texture stream. When using more than one depth camera, it is better to stream a 3D mesh or point cloud, although there is a potential problem with network traffic then. When it comes to real time stream encoding, the open source software FFmpeg was used, invoked from Unity3D scripts in its Windows executable form. This program

provides, along with mpeg-4 codecs, the possibility of hardware encoding using Nvidia NV-ENC or Intel QuickSync Video encoders.

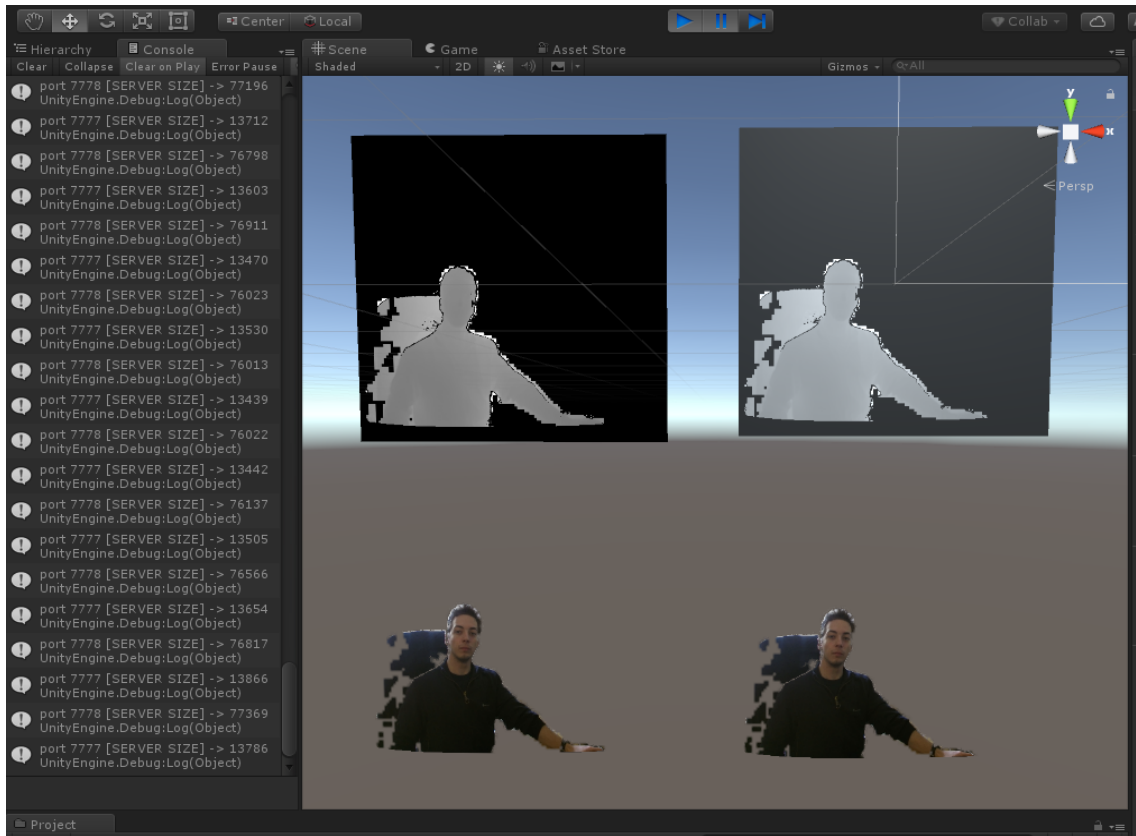


Figure 3.5 – Scene view showing a local networking test with depth and color

3.2 Developed components

Many components developed were replaced by different solutions or were not used in the final version of the system because a different approach was deemed more appropriate. Initially the system was used as shown in section 2 Figure 3.1. The sender component cyclically called its three components; data reading from a source (hard drive, Kinect device, video recording, etc.), data processing (compression, background removal, optimizations, etc.) and data sending (using UNet, a TCP or UDP connection). The receiver worked in a similar way, calling its own three components sequentially: data receiving (UNet, TCP/UDP, hard drive, video, etc.), data processing (decompression,

decoding) and data visualization (point cloud, mesh reconstruction, 2D texture visualization, etc.). This organization of the system was beneficial in the research phases because individual components could be easily swapped out or locked down without influencing the other ones. For example, during network transfer testing, a dummy video recording was used as the data source, with literally no processing done, so that any eventual failures could be attributed to the networking component itself. Similarly, while testing data processing on the receiver side, a 2D quad visualization method was used for the sake of simplicity, as shown in Figure 3.5. The HoloLens device was, however, introduced only in the later stages of development. This changed the approach somewhat, however the knowledge and methods obtained during research proved to be very valuable nevertheless.

Several options were considered for the implementation of network communication between clients. The Unity3D engine offers built in networking in two forms, the low-level API and the high-level API (also known as UNet) which are interchangeable and offer some pre-built network data transfer solutions. Scripts were written using the low-level API to test the ability to transfer large amounts of data in real time. Used in conjunction with depth and color frames, these amounts correspond to about 512x424x4 bytes transferred 30 times per second, not including the audio stream, as the worst case scenario. This approach, however, didn't produce the required bandwidth traffic, and thus Unity engine's built in networking was given up on.

With the constraints imposed by the requirements of the project and the platforms used, the simplest approach was to develop a network communication component from the ground up, using C# native networking libraries which are present in both x86 and Universal Windows Platform environments. It was decided that a reliable TCP over IP was the best solution to preserve the integrity of the data without compromising speed or latency. The network communication scripts use a predefined header to communicate the current frame's length in bytes and other necessary data.

The implementation of this transport protocol, as well as the depth and color multiplexing is described in section 4 of this thesis.

3.2.1 Networking

During development several methods of networking were attempted. One of the first attempts was using Unity's **built-in networking**, however it was unfit for the purposes of this system. One of the problems with the built-in system is that it is difficult to adapt to high bandwidth communication with a requirement for reliable, sequenced data transfer in real time. The large data structures had to be broken up to be sent in fragments using the *ReliableSequenced* or *ReliableFragmented* channel configurations. This led to the need for multiple uses of array copying and preparing functions which further slowed down the processing. Since much of the functionality contained within Unity3D's networking is encapsulated, it is sometimes hard to identify the point of failure when unexpected complications occur. These reasons lead to the abandonment of this method.

The second idea was using parts of the initial framework which uses C++ dynamic link libraries for additional functionality. A network data sending component had been created, using **Boost libraries** for C++. Due to the complexity of asynchronous communication with Boost, and the difficulties concerning data transfer between Unity3D and the dynamic libraries, working with large amounts of data requires careful use of buffers, which are difficult to synchronize when working with data coming from Unity.

One of the considered options was to use **Media Foundation**, a framework compatible with the Universal Windows Platform which is run by the HoloLens device. Another advantage of the Media Foundation platform is built in support for hardware encoders such as the Intel Quick Sync video encoder. However this relies on the FFmpeg application for color and depth image processing (here we are considering that the depth frame is encoded

as a bitmap). This is why when bitmap depth frames were abandoned, this method was abandoned as well.

The ultimate solution was much simpler, a TCP server and client for reliable sequenced communication written in C++ using the native System.Net library. Data is read from a network stream using a **TCP socket** for byte transfer. This code was optimized in later stages of the project. Optimizations of the system will be described in section 4.1.2. This was mostly inevitable due to the blocking functions used in network transfer. It is important to notice that since this network transfer solution works with byte arrays, it is invariant of the type of data which is sent, meaning that the same component works for mesh, audio and color data.

3.2.2 Depth data processing

When raw data is captured on the Kinect V2 sensor, it is obtainable from the Unity plugin SDK. This data needs to be processed before it can be displayed on the other end of the system. Processing includes background removal, mesh construction and data compression. Due to the fact that the system is localized on two devices, each of these three tasks may either be performed on the server or on the client side. Because the HoloLens is a portable device with lower performance than a desktop computer, most of the work was intended to be delegated to the server. For background removal and mesh construction, samples from the Unity3D Kinect v2 samples pack were used as a starting point. This code is fast enough and the quality is satisfactory, both when it comes to background removal and mesh creation. In the later phases, however, it had to be optimized further, in order to ensure that all components work well with real time processing.

One course of development involved using the **FFmpeg executable** to stream the data as an MP4 video stream via a direct connection from the server to the client. This proved to be difficult to implement due to the fact that the program interface is scarcely documented and the errors are non-descriptive. The reason this method was abandoned is that Microsoft

HoloLens runs on the Universal Windows Platform, which is not compatible with x86 architecture built binaries and libraries, which FFmpeg uses. The FFmpeg executable is a console application which takes parameters telling it which data should be processed and how it should be handled. Several Unity scripts had been written, invoking the external program to research feasible communication methods. The main problem was finding a way to send the large amounts of data to the external executable and receive the processed data in return. This was attempted in several ways which, most notably, included passing the data written on the disk (a very slow method not suitable for real-time applications but good for development) and passing the data through a pipeline [7]. The executable takes several parameters, usually these are: source and destination of the data, data formats and format details, image or sample size, used codecs, etc. The input and output can be specified as either files on the disk, sequences of files, web data streams or pipelines. Most widespread formats and codecs are also supported. One of the problems with using FFmpeg with grayscale represented depth frames is that an 8 bit grayscale image reduces the precision of the depth data, as it is conventionally stored in 11 bits. Using an unconventional 16 bit grayscale image is a bit more difficult as Unity3D does not support such formats.

When it comes to the implementation of the Unity3D to FFmpeg executable communication there are several things to address. The FFmpeg process is started from the script using functions from the *Systems.Diagnostics* namespace, with the following *StartInfo* settings: *UseShellExecute* is false, *CreateNoWindow* is true, and we have redirected the standard input, the standard output and the standard error of the process. The executable uses the standard error stream to show info to the user, and since we want no window to be displayed, the standard error stream should be redirected and read. Displaying it with *Debug.Log* gives us the only insight into what is happening within the FFmpeg program if processing fails. Input is given to the process via the standard input and output is read from its standard output. There was a lot of experimentation with arguments used for starting the program [5]. Let's see an example, starting FFmpeg.exe to

transcode a JPG image to an MP4 video (with one frame). Entering the following line into the console produces the “test.mp4” output file in the same folder where the input image “input.jpg” is located:

```
ffmpeg.exe -y -i input.jpg -c:v h264 test.mp4
```

The “-y” argument tells the program to overwrite any output files without asking for permission. The “-i” denotes following input file(s) and here it is possible to provide the image size or the format with the “-pix_fmt” option. Then, the “-c:v” option is short for “codec:video” and tells the program which codec should be used for transcoding. Finally, the output file is specified. Similarly it is possible to use pipelines as input or output by writing “pipe:1”, for example.

Eventually several scripts were written to convert the depth data into a **grayscale bitmap** with 24 bits per pixel, all three channels (red, green and blue) populated by the same value for the purpose of testing. Although this might sound like a terrible waste of resources, these bitmaps can actually be compressed very efficiently, as if only one channel was used. They are easily converted into byte arrays in Unity, which are then sent as a stream over the network. Byte arrays can also be easily compressed in real time. For this, C#'s **Deflate class** was used, as it provides quick byte stream compression which is suitable for real-time applications. The Deflate algorithm found in the namespace *System.IO.Compression* creates a *DeflateStream* from an underlying *MemoryStream*, which is suitable for real time compression. It is a lossless data compression algorithm [8] which uses LZ77 and Huffman coding. The algorithm takes a byte array and replaces duplicate sequences with pointers, and replaces normal symbols with weighted symbols based on frequency of use. Specifically, Huffman coding replaces common longer byte sequences with their short identifiers which take up less space in the stream. The LZ77 algorithm replaces repeating sequences with a single copy, plus a *length-distance pair* which denotes how many of the following characters are equal to *distance* characters behind it.

Before sending the user's mesh over the network, the mesh needs to be serialized; converted into a byte array. For that purpose the system uses a mesh serializer, which writes the individual mesh components (mesh header, vertices, triangle indices, and UVs) to a data stream which is then saved to a byte array. A complementary functionality is also contained within the mesh serializer, so that deserialization could be done on the receiver. Since this is a relatively programmatically simple (yet quite resource consuming) operation, the same class could be used in the Windows Holographic build with little adjustments. In the optimization phase of the sender development, this class was revised so that the serialization could be parallelized. This meant replacing the Unity's non thread-safe *Mesh* class used in the serializer with a custom *MeshThreadsafe* class which is basically just a holder for the data of interest within the mesh itself.

3.2.3 Color data processing

Color data read from the Kinect V2 sensor can be either in full HD (1920x1080 pixels) or in the same resolution as the depth image, which simplifies texture mapping. It also reduces the network traffic of the system greatly, due to the huge difference in data size of each frame. Most of the methods used with depth image bitmaps work the same with color bitmaps, so finding fast and efficient compression and encoding algorithms was an important focus. Color data, however, due to the fact that it uses four channels (32 bits, RGBA) instead of two used by depth data, generally takes up a bigger percentage of the network traffic. This is the first reason why using the full HD texture stream was eventually abandoned in favor of the smaller depth resolution texture. The second reason is that using the full HD texture requires additional processing to map the texture correctly to the depth image. This is a consequence of the fact that the 1920x1080 resolution image not only has a different aspect ratio than the 512x424 pixel resolution image, but also has a different field of view.

A problematic segment of the color texture obtaining script is conversion from a graphical memory stored `RenderTexture` to a RAM stored `Texture2D`, and it cannot be parallelized because the `ReadPixels` function is a non thread-safe Unity3D API function. The following code demonstrates the problematic section of the `SourceColor` script.

```
renderTexture =
userMeshRenderer.material.GetTexture("_MainTex") as
RenderTexture;

RenderTexture.active = renderTexture;

// Copy data from GPU to RAM

finalTexture.ReadPixels(new Rect(0, 0,
renderTexture.width, renderTexture.height), 0, 0);
```

The call to `GetTexture` obtains a `RenderTexture` from the main renderer which uses the texture from the `KinectManager` and the underlying API, which is processed in the GPU memory. The texture has to be locked in the GPU so that it could be safely copied to the RAM, which is a time-expensive operation because data cannot be copied both ways simultaneously.

Additionally, a **delta-encoding** script was created to work with general byte arrays, in order to maximize the compression. Delta encoding works by storing data in the form of full and *delta* (difference) frames. Frames are recorded in a fixed interval and stored as a difference relative to the full frame which is refreshed periodically. This greatly reduces redundancy, since adjacent frames often share much of their data. Instead of using an existing delta compression library, a custom-tailored component has been developed to best fit the requirements of this project. This component will be described further in section four.

The sender-side color processing script includes handling the loaded texture, delta-encoding and compression of the data, as well as supplying the data to the network sender component.

3.2.4 Other components

For the purpose of data sender project optimization a script was written for **threading efficiency testing**. The script relies on tested functions to report when they have reached a section which was determined problematic (taking up a lot of time to be completed) and when they have left it. Each of the threads calls a function of the *ThreadDurationTest*, as it is called, to obtain a unique ID which will later be used to identify it. As a thread calls the *WriteCheckpoint* function of the duration test script, a unique timestamp is written to a file, identifying the start of the “critical” section or its end, if a start was previously written. Once a file is produced, this data can be processed and visualized. For that, we have developed a processor script which is written in the Python language and uses several libraries from the SciPy packages. It reads the checkpoint file and draws a graph from the data. An example is shown in Figure 3.6. The different colors represent different threads, and the spikes denote thread activity in the “critical” section of

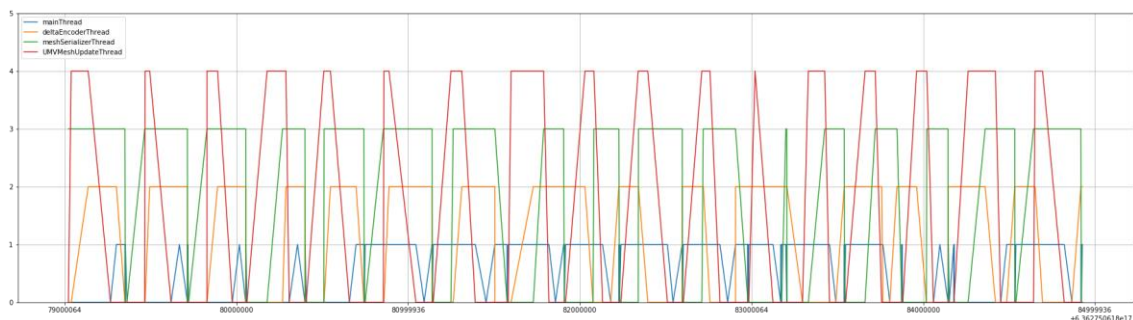


Figure 2.6 – Thread activity graph

interest. Similar graphs were used to determine which parts of code can be parallelized and which parts the most important. The most time-consuming processing was segregated into a new thread or further optimizations were made to the code.

A session recorder has also been developed to record the user mesh and texture data in a broadcasting session. This was found to be useful for receiver-side testing and demonstrations. This recorder is further described in section 4.1.1.

Audio data is also streamed over the network, captured on the Kinect's microphones. An *AudioClip* is created so that it could be used for audio capture from the microphone, as shown in the following line of code:

```
audioSource.clip = Microphone.Start(microphoneDevice,  
true, 1, recordFrequency);
```

. Recording is initiated with the *Microphone.Start* function of Unity's API and the result is saved in our audio clip. Raw audio data is obtained from the clip with the *GetData* function and saved in a buffer, which is later compressed into a byte array (using *VoiceUtils.Compress*) and sent to the network script. A circular buffer is used in the script to hold audio data frames between sending intervals.

4. Developed system

This chapter covers the methods used in the communications platform, for each of the previously discussed segments of the system: depth data representation, background removal, mesh creation, data compression, networking solutions and data displaying. One of the goals when building the system was to keep as much processing as possible localized on the server, so that the HoloLens device would handle only receiving and displaying the data. The diagram below shows the basic operation scheme of the system. These are the main components of the system and their order of execution.

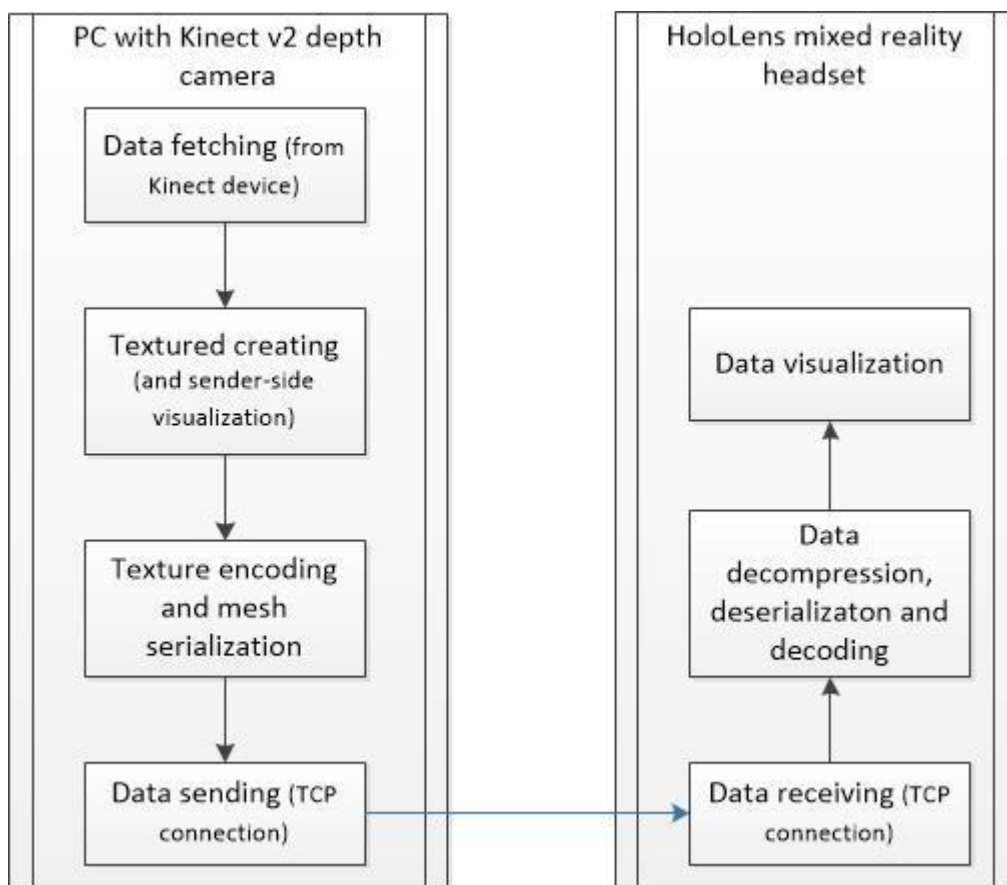


Figure 4.1 – System component scheme

4.1 Video and audio data sender

A very important part of the sender is the *UserMeshVisualizer* script taken from the *KinectUserVisualizer* scene of the Unity3D Kinect V2 SDK Samples which can be found on the Asset Store. This script was modified and parallelized to optimize its performance and to adapt it to this use case. It takes the color and depth images and creates a 3d textured mesh of the user with the background removed.

The transparent texture is fetched from the background removal manager and converted from a RenderTexture to a Texture2D using the already described procedure. This is all done in the main thread since the copying cannot be parallelized, unlike the mesh creation, runs in a separate thread. The previous mesh is cleared, and the newly computed vertices, uv array and triangles are assigned to the mesh that is being displayed in the scene. Thread synchronization is done by means of flags indicating when the created mesh is being copied from the buffer into the mesh used for network sending. Mesh updating is done in a function which passes through the whole depth frame sampling every *sampleSize* point (1, 2, 4 or 8, see section 4.1.2 for mesh quality settings) and creates triangles from adjacent vertices. To reduce the size of buffer arrays allocated on each execution of the mesh creation function, the size of the arrays is estimated based on the distance between adjacent points. If a point is much farther away from the depth camera on the Z axis, which means it is probably not a part of the user's point cloud. Also, points with the maximum distance (or distance zero) are automatically discarded as well. The function is time-expensive due to the numerous passes through the 512x424 point sized array.

4.1.1 Sender components

Data representation

In the developed system, the depth data stream read from the sensor is interpreted as a point cloud. This allows expansion in terms of more depth

sensors used for a more complete and accurate 3D representation and enables the server to handle the computation-heavy processing related to mesh construction.

Data compression

There are two phases involved in reducing the amount of network traffic. The first is the **delta-encoding** of byte arrays, which works with general byte arrays, on the principle of sending a full frame and then sending only the difference byte array of the same length. The *deltaEncode* function receives three arguments; the serialized texture (a byte array), a flag telling whether the current frame will be a full frame or a delta frame and a desired loss percentage. If a *full frame* is about to be sent, the received texture is saved for the next calls of the function where it will be used to compute delta frames. Otherwise, the *delta frame* is computed by subtracting the previous frame from the current texture. Figure 4.1 shows a full frame and a delta frame following it.

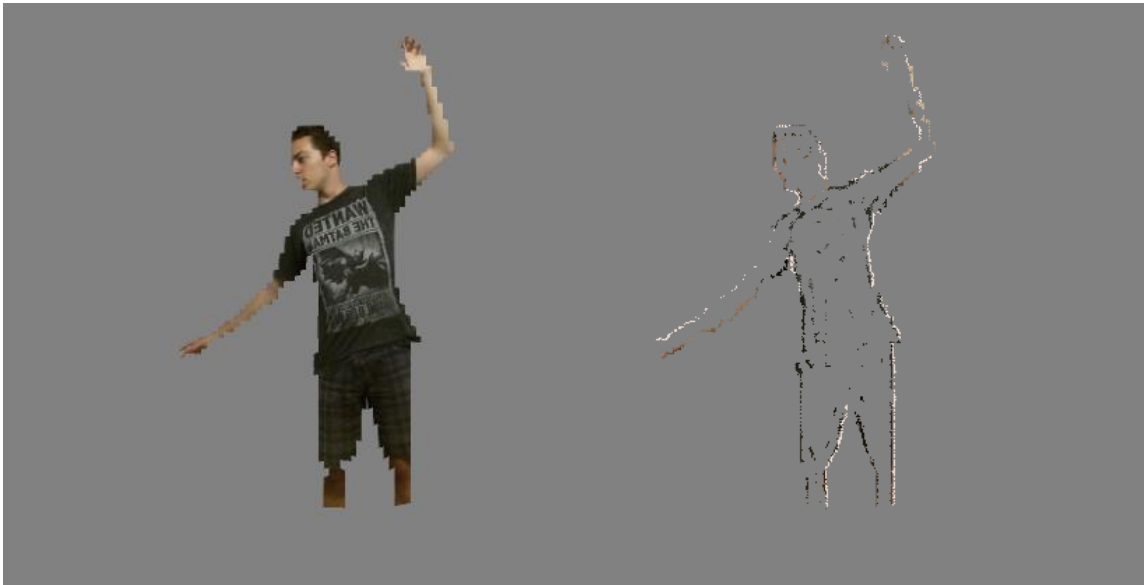


Figure 3.2 – Full frame and delta frame

The delta frame is then checked against the previous frame byte by byte, with each byte of the delta being equalized to the previous frame's byte if the difference is less than the minimum difference allowed by the given loss factor. This frame is then returned to be processed and sent to the receiver. Similarly the decoding algorithm just adds the previous frame to the received data to obtain a full frame on the receiver side. This leaves very little processing to the portable HoloLens device.

Although each of the sent dataframes has the same length and no size reduction is apparent, the effects become obvious when delta encoding is combined with a byte array compression algorithm, such as **Deflate compression**. Since most of the frames sent (all *delta* frames) only mark the difference between the previous and the current frame, most are filled with zeros, because most parts of the byte array are unaffected between adjacent frames. This allows for high compression factors, while keeping the compression lossless [9].

The delta encoder developed for this system uses a timer to determine when a full frame needs to be sent instead of a delta frame. It was found to be appropriate to set the full frame interval to 3 seconds, when using the system in a local area network.

Network data transfer

To ensure all data was transferred correctly (it would be destructive if a descriptor or header was sent incorrectly or received partially) and to preserve the integrity of the client-side system [8], a **reliable TCP** connection was used between the server and the client. The TCP socket works on three connections, using different ports for each type of data. Therefore port 7777 is reserved for color data, 7778 for mesh data and 7779 is for audio data. To enable the client to interpret data correctly, all data had to be packaged together with additional **header data**, describing the content of the network dataframe. This header defined the type of the data sent, color or mesh, the size of the network frame to be read from the network stream as well as the size of the decompressed dataframe (which happens to be a constant value when it comes to color data, exactly 512x424x4 bytes per color frame).

The message preparation function in the *SenderTCP* script takes two arguments; the length of the inflated message and the data byte array itself. First, data is compressed using the Deflate algorithm. The length of the compressed data is saved and size of the entire message is computed as the compressed data length plus the compressed data header which tells the receiver how big the inflated data array is. Afterwards the network packet array is packed in the following order: first the size of the entire message (so that the receiver knows how many bytes to read from the stream), then the size of the inflated data array and then the compressed data itself. Figure 4.2 shows the network message contents.

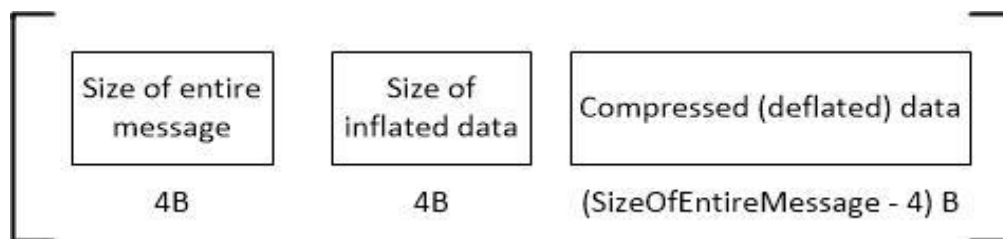


Figure 4.3 – Network message header components

Data recording

One of the features of the server side system is the recorder. This allows for playback of the depth, color and audio data which can be recorded during a session. The recorder uses data type flag to tell the receiver whether the packet received contains depth or color data. A timestamp is also written next to each frame, allowing the playback system to know when to play a frame, keeping the recording real-time. This is necessary due to the fact that spikes occur in the framerate, making it impossible to save frames in a regular interval.

The recorder uses a script similar to the network sender, packing data with a header which is then streamed into a recording file. The data recorder's *PrepareMessage* function takes four arguments: size of inflated data array, the actual data array, a timestamp and a flag telling the data type (color or mesh). Figure 4.3 below visualizes the structure of the recording packet.

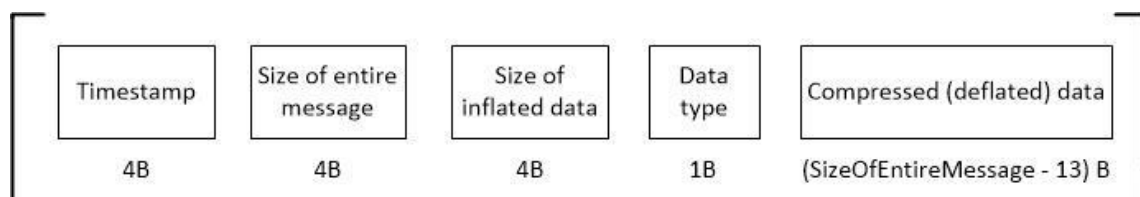


Figure 4.4 – Recorder frame header contents

4.1.2. Sender side optimization

When many of the server side components are run together, performance becomes a problem even though the hardware can be quite powerful. This is why the program needs to be well optimized, multithreaded and smartly organized. The application was decomposed into parts to see which ones can be parallelized. Some parts of the program must be executed sequentially, for example data must be first obtained, then processed and

finally sent. This is not trivial to parallelize. The system works in multiple threads, with the main thread performing all the Unity3D related processing (which cannot be parallelized as most Unity3D methods aren't thread safe) such as mesh creation, and other threads working on individual tasks such as compression, encoding, networking and data obtaining from the Kinect device. The synchronization of threads is done with simple flags which determine when a critical section is in use by another thread. Figure 2.1 shows thread activities, in a small period of time.

The most important parallelization of the system occurs with the color and depth data read from the Kinect V2 sensor. In one thread, raw depth data is fetched and a mesh is created, to be converted into a Unity3D Mesh in the main thread. It is then serialized, delta encoded and compressed. Another thread takes care of the texture, converting the RenderTexture from the sensor API to a Texture2D, serializing it, delta encoding and compressing the data. The conversion from a RenderTexture to a Texture2D is an important part of code to be optimized, because it requires the texture to be copied from the GPU memory to the RAM, which can be costly as there is a bottleneck when simultaneously reading from the GPU and writing to it (the Kinect V2 drivers write to the texture). There is yet another thread tasked with adding headers to data frames and sending them over the TCP connection, as the network functions are blocking until the client reads them. Figure 4.4 shows the final parallelization diagram of the system.

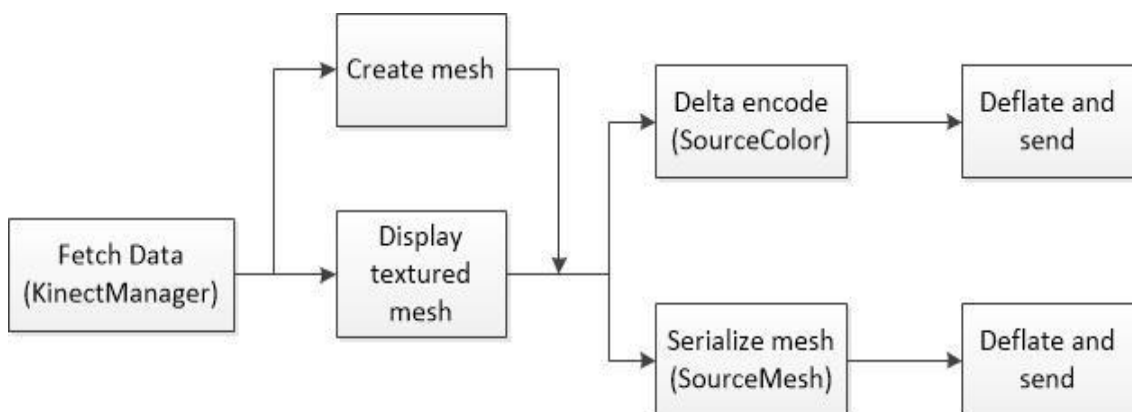


Figure 4.5 – Sender-side parallelization diagram

First, the raw depth and color data is obtained from the Kinect Manager, which provides an API to access raw and processed data from the Kinect V2

device in Unity3D. This data is then fed into the mesh construction code in the *UserMeshVisualizer* script, while at the same time, the main thread displays the created mesh with the texture applied to it. The created textured mesh of the recorded user is already filtered, the background is removed from the data and thus it is ready to be sent over the network. However, it has to be processed before it can be sent to the receiver side. Since the mesh and color components are sent separately, they can also be processed separately. Therefore two threads perform mesh and color processing in parallel. The color texture is serialized into a byte array before being delta encoded. After the encoder it is fed into the deflate compressor and sent over the TCP connection. Meanwhile, the mesh is serialized (it is a time-consuming operation, just as is delta encoding) and, just like the color data, fed into the compressor before being sent to the receiver.

All these steps, along with optimization of the individual components, ensure that the server can deliver the data to the HoloLens device in real time. There are several options which can reduce the network traffic or the processing complexity by sacrificing quality. These options can be set when the server program is run, in the Setup scene. Image 4.5 below shows the sender-side menu screen.

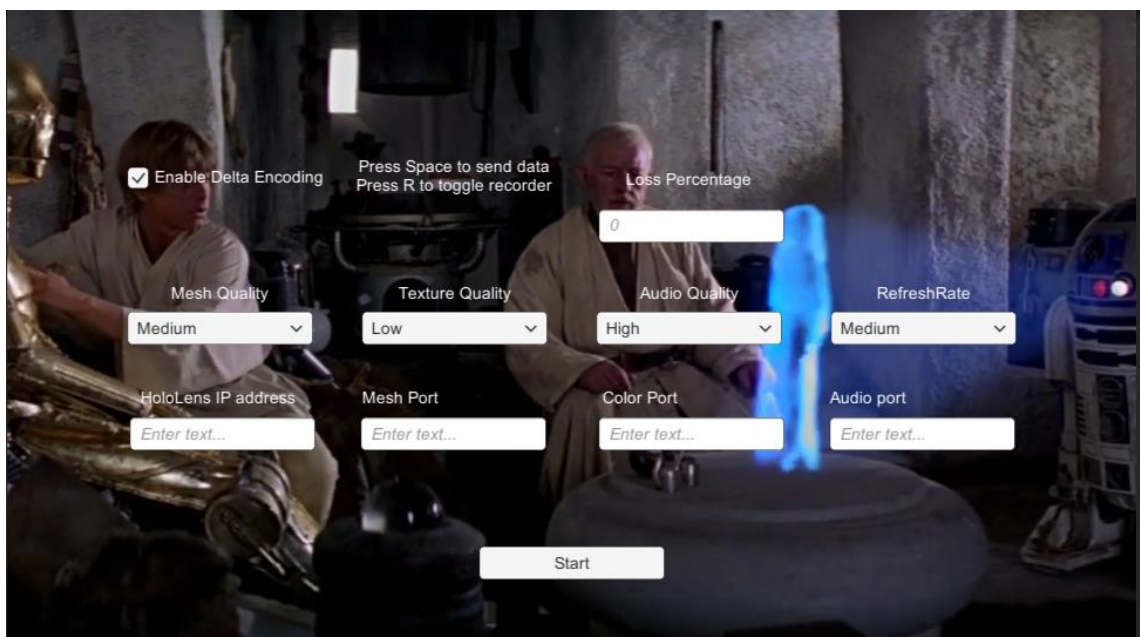


Figure 4.6 – Setup scene menu screen

Mesh quality can be selected between the Low, Medium, High and Unstable High settings, which are manifested by the sample size in the mesh creation script. Lower the quality and less depth points will be sampled when the mesh is created. For the Unstable High setting, every depth point is taken, for the High setting, every second is taken, for the Medium setting every fourth and so on. This causes the vertex count in the mesh to vary greatly, and thus reduces not only network traffic but processing complexity at the expense of quality. The Unstable High setting may cause the mesh to break due to a constraint in the Unity3D engine stating that the maximum vertex count of one mesh can be 65 000, hence the name.

Another quality setting is the **send interval**, which can be set to send data to the receiver at either 15, 24 or 30 times per second. Delta encoding can be set to incorporate a **loss factor**, which can also be changed in the menu. This allows to redefine the minimum amount of pixel difference between two adjacent frames in order to reduce the amount of data populated delta frames. The loss in quality is, however, quite obvious when using larger loss percentages. It was also planned to incorporate texture quality selection, due to the fact that the Kinect V2 sensor provides a full-HD RGB image. This



Figure 4.7 – Mapping the full HD texture to the depth frame produces an offset in the mesh texture

resolution brings several problems when it comes to mapping the texture to a 512x424 resolution depth image, because its aspect ratio, field of view and other parameters are different, and for these reasons this feature was ultimately not implemented. Image 4.6 visualizes the problems related with mapping the full HD texture to the depth frame.

Measuring the performance of the data sender process is certainly not a very simple task. Apart from depending greatly on the hardware that runs the program, performance also varies on many factors, such as room lighting, user's distance to the depth camera and his position as well. Certain measurements have been made despite these factors, to compare between the mesh quality settings and to see how they influence the framerate. The test machine runs an Intel Core i7-4702MQ 2.2 GHz CPU with 16 GB DDR3 L RAM, and has an NVIDIA GeForce GTX 850M graphics processing unit. On both the *low* and *medium* mesh quality settings the framerate was roughly the same, about 26 frames per second (with the network sender set to send data 24 times per second for all quality settings). On the *high* setting, 21 frames per second was the average, and on the *unstable high* setting this is reduced to only ten frames per second. The performance in the first three settings was primarily influenced by the color texture conversion and obtaining code (it caused the most notable delay in the system) but in the unstable setting the massive framerate drop is caused by the *KinectManager* script, which needs to allocate much more space for data processing. This allocation, together with the need to process much more data, causes large delays, and is one of the reasons it is not recommended to use the *unstable high* setting in the current system build.

4.2 Data receiver and visualizer

This chapter briefly describes the data receiver, run on the Microsoft HoloLens device. The receiver side of the system is not the focus of this thesis, and will therefore be covered with less detail. The base for integrating HoloLens functionality in Unity3D is a collection of scripts and components aimed at simplifying development of holographic applications for Windows

Holographic (the platform run by the HoloLens device). This is supported in Unity3D since version 5.6.1f1 and provides access to HoloLens input management, sharing, spatial management and building applications for the platform. There are three main components in the receiver build scene, so let's describe them one by one.

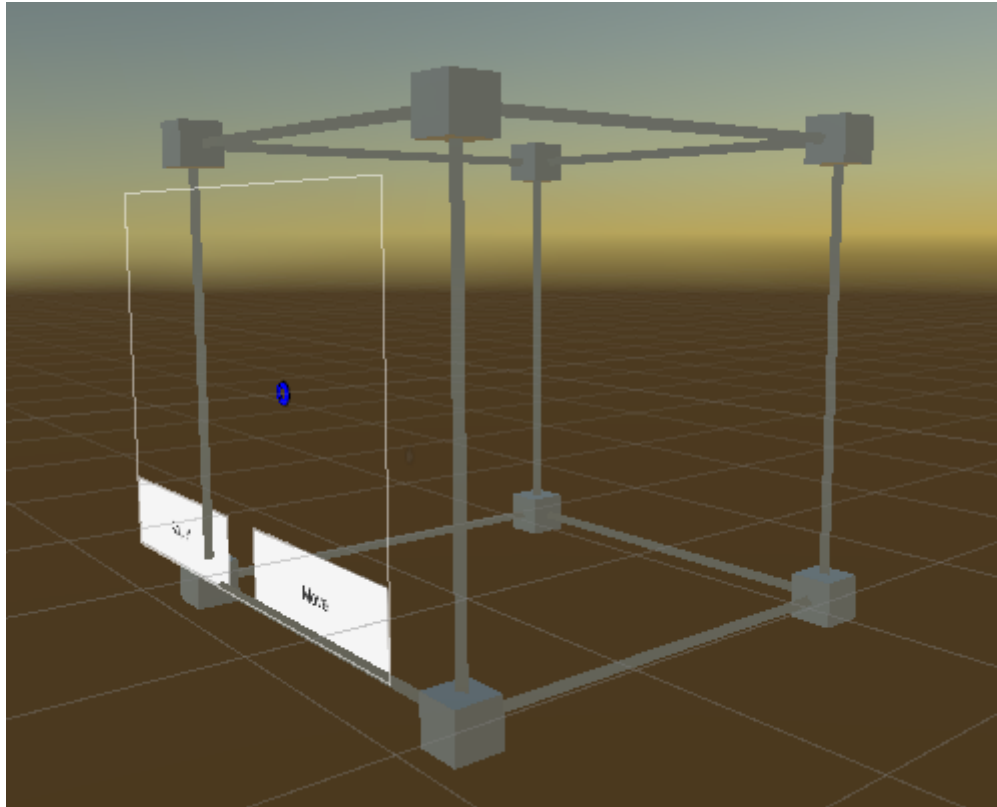


Figure 4.8 – HoloLens Unity3D graphical user interface for object movement and scaling

PlayerHoloTalk is a prefab object used to represent a person in the HoloLens visualization. It allows user interaction with the hologram displayed in mixed reality, more precisely it allows scaling, rotation and movement of the visualization using HoloLens specific commands from the HoloToolkit. Image 4.7 shows the HoloLens UI as seen in the Unity scene view. *PersonHologram* is the object responsible for converting the received byte arrays to the final Mesh and Texture2D color data. This conversion process is similar to the sender's process, and includes inflating (decompressing) the data, deserializing the Mesh and running the color texture through the delta encoder to obtain the original. This object also holds the visualizer script, which uses a buffer to display the data. This buffer saves several frames before visualizing them in real time. The *Network* component is also analogue to the sender's component, receiving data on three ports (7777 for color, 7778 for mesh and 7779 for audio data) via the TCP socket.

HoloLens development in Unity3D is, despite the numerous aids available, still quite a challenging task. Deploying projects to the device is

tedious and takes a lot of time (a few minutes), so it is not possible to make quick tests after changing a small part of code. Windows Holographic does not support some of the libraries and namespaces used in the sender side [10], so two versions of the code need to be developed: one for testing on the desktop computer running Windows (the development machine for the HoloLens) and the other for the Windows Holographic platform (a subset of the Universal Windows Platform). Development is also hampered by the fact that it's impossible to access some useful features of the HoloLens device on a lower level, for example there is no way to access raw depth data (access is only possible through the Spatial Perception component). The camera can only be accessed by one application at any given moment and its live feed is not obtainable (it either has to be saved as a video file or accessed image by image). HoloLens API doesn't enable developers to add new hand gestures [10] or even to access raw hand tracking data. The lack of these functions somewhat restricts the possibilities of the device in conjunction with Unity3D.

Several methods of data visualization were used on the receiver side, most of which were also used on the sender side in various phases of testing and development. The three which saw most use are: point cloud visualization (see Figure 3.2 in section 3.1), mesh visualization (used in the final system because it provided the most realistic and accurate results) and 2D sprite visualization (used in the early phases because of its simplicity and possibility to rule out errors). The image below shows the depth frame rendered with the point cloud method and visualized as a grayscale texture on a 2D quad. The HoloLens device had no performance issues running the decompression, deserialization and mesh visualization because all of the remaining processing was done on the more powerful data provider device. The mesh deserialization still had to be parallelized to maximize performance since HoloLens devices possess a dedicated GPU and a multi-core processor.

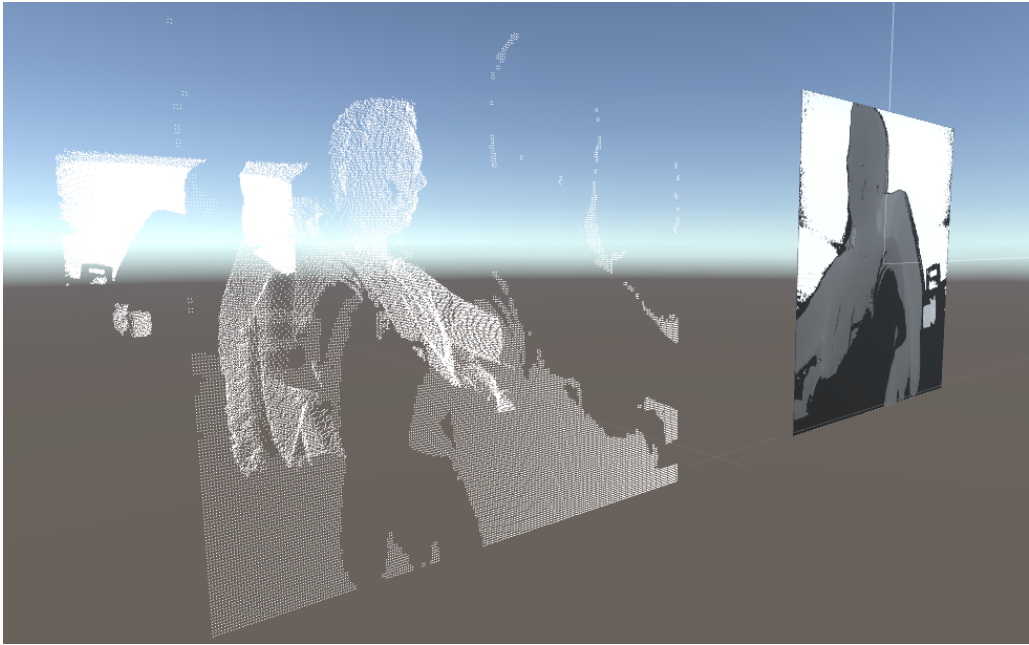


Image 4.9 – Point cloud and 2D texture methods of visualization

5. Conclusion

This system provides a proof of concept and a technology demonstration for real time holographic communication. With some additional optimizations it will be possible to implement two way communication and one way communication with multiple receivers (broadcasting; which was the basic intended purpose of the system). Many of the solutions found during various research phases of the project may prove useful in similar systems even if they were deemed unsuitable for this project. The system can also be upgraded to use multiple depth cameras for a more accurate representation of the recorded person, to give the users a better feel of presence. With multiple depth cameras recording from all angles, a full 3D model of a person could be rendered in real time on the receiver side, unlike the current mesh produced by one camera, as shown in Figure 5.1. More complex mesh reconstruction methods [11] which use machine learning can also be implemented to further reduce network traffic and make the application more accessible. To show more detail of the user's face, his mesh may be split into two parts with the head having a more accurate depth sampling and the body, which requires less detail, being subsampled in lower quality. Finally, although the system was built for current generation depth cameras and holographic headsets, different hardware can be used to provide more accurate visualizations or application-specific advantages. Even if the system is revamped to an entirely different configuration, for example using a mobile phone instead of a holographic headset or a RGB camera for background removal instead of a depth camera, many developed components can be reused with some modifications.

This thesis gives a high level overview of the methods that were considered to solve individual problems encountered in designing a real time communication platform, as well as the solutions that have been created as a result of the research. It sums up the difficulties that have been encountered as well as their solutions in the form of concrete components designed around the Unity3D engine. The focus of the thesis is on used methods rather than on the specific components.

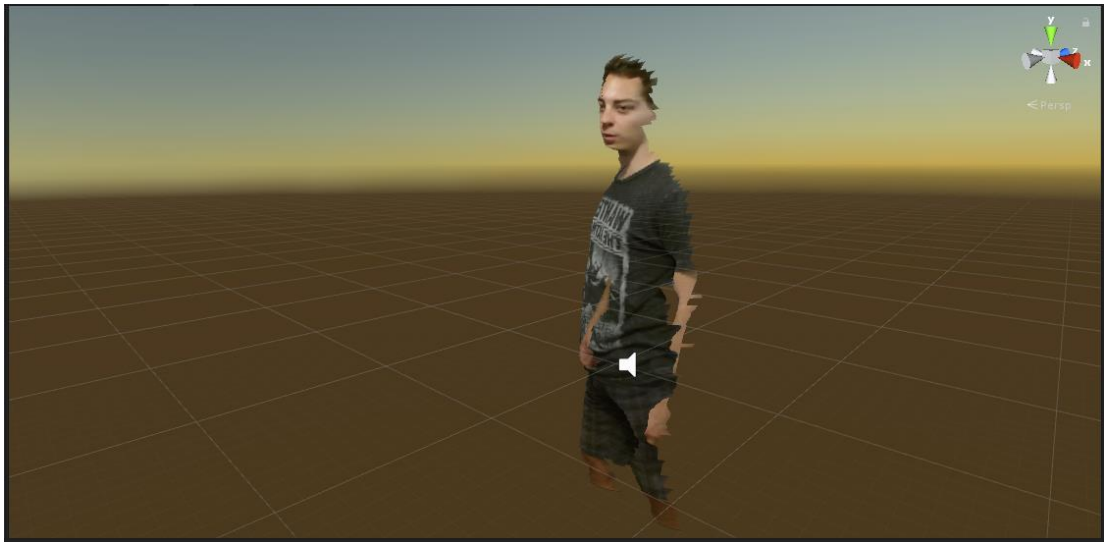


Figure 5.1 – 3D user mesh visualization viewed from the side

Sažetak

Holoportacija modela čovjeka

U ovom radu implementiran je sustav za udaljenu jednosmjernu mrežnu komunikaciju u stvarnom vremenu koristeći prijenos holografske slike koji uključuje prijenos modela, teksture i zvuka. Koristeći grafički pogon Unity3D kao podlogu, razvijen je sustav koji koristi Microsoft Kinect v2 i HoloLens uređaje za prijenos podataka preko mreže od poslužitelja prema klijentu. Ovaj sustav predstavlja dokaz koncepta holografske komunikacijske platforme. Rad također pokriva razvijene koncepte i komponente, i pruža pregled raznih faza razvoja projekta kako bi pokazao kako je konačan sustav oblikovan.

Ključne riječi: miješana stvarnost, holoportacija, hologram čovjeka, komunikacija, stvarno vrijeme, Unity3D, C#, dubinska kamera

Abstract

Holoportation of human models

This thesis implements a system for remote, end-to-end, one way, real time holographic streaming, which includes audio, mesh and texture streams. Using the Unity3D engine as the base, a system was developed, using the Microsoft Kinect v2 and HoloLens devices for server to client streaming over the network. This system provides a proof of concept for a holographic communication software platform. The thesis also covers the developed concepts and components, and provides an overview of the various stages of development to demonstrate how the final system was formed.

Keywords: mixed reality, holoportation, human hologram, communication, real time, Unity3D, C#, depth camera

Bibliography

[1] Microsoft Developer Network, "Kinect V2 Preview SDK now available includes UNITY3D plugin",

https://blogs.msdn.microsoft.com/uk_faculty_connection/2014/07/17/kinect-v2-preview-sdk-now-available-includes-unity3d-plugin/, July 2014

[2] HoloLens hardware details, Microsoft Windows Dev Center, https://developer.microsoft.com/en-us/windows/mixed-reality/hololens_hardware_details

[3] E. Lachat, H. Macher, M.-A. Mittet, T. Landes, P. Grussenmeyer, "First experiences with Kinect v2 sensor for close range 3D modeling", The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-5/W4, 2015 3D Virtual Reconstruction and Visualization of Complex Architectures, Avila, Spain, 25-27 February 2015

[4] Radu Bogdan Rusu ; Steve Cousins, "3D is here: Point Cloud Library (PCL)", Robotics and Automation (ICRA), 2011 IEEE International Conference on, 2011

[5] ffmpeg Documentation, FFmpeg Web site, FFmpeg.org, May, 2017,

<https://www.ffmpeg.org/ffmpeg.html>

[6] Dapeng Oliver Wu, University of Florida, "FFmpeg real time encoding/decoding, x264 codec",

<http://www.wu.ece.ufl.edu/projects/wirelessVideo/project/realTimeCoding/download/doc/howto.pdf>

[7] Pipes, Interprocess Communication, Microsoft Windows Dev Center, [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365780\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365780(v=vs.85).aspx)

[8] Wai-Tian Tan ; A. Zakhor, "Real-time Internet video using error resilient scalable compression and TCP-friendly transport protocol", IEEE Transactions on Multimedia, vol. 1, No. 2, pages 172-186, June 1999

[9] S. Mehrotra, "Lossless depth frame coding", Microsoft Research Redmod, 2011, <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/depthcode-final.pdf>

[10] Windows Mixed Reality Developer Forum,
<https://forums.hololens.com>, 2017

[11] Remondino Fabio, "From point cloud to surface: the modeling and visualization problem", International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV-5/W10, January 2003.