DynCam: A Reactive Multithreaded Pipeline Library for 3D Telepresence in VR

Christoph Schröder
University of Bremen
Bremen, Germany
schroeder.c@cs.uni-bremen.de

Mayank Sharma
CERN
Geneva, Switzerland
mayank.sharma@cern.ch

Jörn Teuber
University of Bremen
Bremen, Germany
jteuber@cs.uni-bremen.de

Rene Weller
University of Bremen
Bremen, Germany
weller@cs.uni-bremen.de

Gabriel Zachmann
University of Bremen
Bremen, Germany
zach@cs.uni-bremen.de

ABSTRACT

We contribute a new library, DynCam, for real-time, low latency, streaming point cloud processing with a special focus on telepresence in VR. Our library combines several RGBD-images from multiple distributed sources to a single point cloud and transfers it through a network. This processing is organized as a pipeline that supports implicit multithreading. The pipeline uses functional reactive programming to describe transformations on the data in a declarative way. In contrast to previous libraries, DynCam is platform independent, modular and lightweight. This makes it easy to extend and allows easy integration into existing applications. We have prototypically implemented a telepresence application in the Unreal Engine. Our results show that DynCam outperforms competing libraries concerning latency as well as network traffic.

KEYWORDS

Collaborative Distributed VR, Functional Reactive Programming, Point Clouds, RGBD Streaming, Latency Measurement

ACM Reference Format:
Christoph Schröder, Mayank Sharma, Jörn Teuber, Rene Weller, and Gabriel Zachmann. 2018. DynCam: A Reactive Multithreaded Pipeline Library for 3D Telepresence in VR. In *Proceedings of Virtual Reality International Conference (VRIC'18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Collaborative virtual environments (CVEs) play an important role in a world where more and more decisions are made based on virtual prototypes and simulations. Designs for new cars, manufacturing plants, and aircraft are no longer just two-dimensional blueprints but are directly created in 3D. Experts meet to make important design and manufacturing decisions. Current systems either use 2D displays or 3D powerwalls to provide each participant a shared view of the 3D data. Both methods have several drawbacks: the shared 2D views limit the perception of depth. Powerwalls support stereoscopic view of the scene, however, usually, only a single user's head is tracked. This can lead to confusion when pointing in 3D space [14]. There are first experiments with multi-user powerwalls but they are only very recently becoming available, and they are restricted to a very small number (up to 6 in lab prototypes) head-tracked users [14]. Furthermore, both methods have a big limitation in common: All participants have to be in the same physical location which can require time-consuming traveling.

Distributed CVEs can be a viable solution in many cases. Especially because the advent of high-quality consumer VR headsets makes distributed CVEs appealing. However, a realistic representation of other participants in such a CVE is essential for communication and collaboration. Currently existing distributed CVEs usually show only parts of the body, like the tracked hand controllers, or they use non-personalized avatars with skeletal tracking. Current research has shown that the quality of the avatar directly influences behavior and the team performance [29]. Personalized avatars significantly improve virtual presence and virtual body ownership as they are closer to realism compared to generic counterparts [30]. However, creating personalized avatars is relatively time-consuming, complex and requires pre-processing [1]. Moreover, even such personalized avatars can be problematic in case of design reviews where the pointing accuracy is essential. Waltermate et al. [30] also imply that higher degree of virtual body ownership achieved by avatars that are closer to reality, ultimately leads to better immersion in virtual environments. The most realistic avatar, i.e., a real-time representation of a person, would therefore result in an optimum immersible experience.

We propose to use a different approach to overcome these drawbacks: our goal is to directly use the RGBD data delivered by depth cameras that are usually used for skeletal tracking. These RGBD images can be used to create a point cloud of the user that we directly use for rendering. This method automatically leads to completely personalized avatars for each participant in the CVE and simultaneously gives strong perceived ownership over the avatar. However, there are several challenges to solve: First, we have to take care of occlusions that occur from persons and objects in front of the RGBD camera. This can be solved by using not a single, but multiple RGBD sensors for each user. The data rate of modern RGBD sensors is relatively high. Therefore, we have to consider methods

VRIC'18, April 2018, Laval, France

ACM ISBN 978-1-4503-5381-6…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to compress the data before sending it through the network to the remote participants in the distributed CVE. Additionally, in real-time environments, latency is also an issue. Finally, the processed RGBD images have to be rendered.

In this paper, we present a system that fulfills all the requirements mentioned above. The core of the system is our novel software framework, called DynCam, for real-time processing of streaming RGBD data. The main functionality of DynCam is to gather streaming RGBD images from one or several depth sensors, to combine them to a single point cloud, compress the data and transport it through networks with the lowest possible latency. In a typical distributed CVE scenario, there are many camera clients that record and pre-process RGBD data from connected cameras and send them to a central server. At the server, the different clouds are further processed and transformed into a common coordinate system. The final point cloud is transmitted back to the clients for visualization. To handle such large amounts of data flexibly and in real-time, DynCam supports multithreading through the use of *reactive programming*. In contrast to other point cloud-oriented libraries, like ROS, DynCam is platform independent, modular and lightweight. The rendering of the point cloud is not part of DynCam itself to retain its platform independence. We show a telepresence prototype where DynCam is integrated into the Unreal Engine and use the rendering of splats to display the point clouds. Further, we compared DynCam to competing point cloud processing libraries that support at least network transfer of streaming RGBD data, even though DynCam offers much more functionality like its telepresence-oriented pre-processing. To do that we have optimized the latency measurement proposed by [25] and, additionally, we consider the network load. Our results show that DynCam receives much better latency while reducing the network traffic thanks to supporting compression.

2 RELATED WORK

Current 3D telepresence systems for distributed CVEs either skip the representation of user's body or represent it through avatars or through 3D visualization of HMDs and controllers in the virtual environments. The hardware capabilities of such systems play a crucial role in conveying telepresence. HMDs like Samsung GearVR, LG 360 VR, and variants of Google Cardboard rely on the Inertial Measurement Units (IMUs) installed in mobile phones to track the orientation of the user's head [3]. Recently, eye gaze tracking cameras installed in HMDs (e.g., Samsung ExynosVR III) allow systems to utilize corneal position tracking algorithms [28] to enhance the User Experience (UX) in virtual environments. More sophisticated systems based on the HTC Vive and Oculus Rift can also track hand-motion controllers using dedicated infrared cameras besides tracking the HMD.

Massively Multiplayer Online Virtual Reality (MMOVR) games like OrbusVR [8] offer a good proof of concept for telepresence-enabled VR games. These games use tracking information available from IR cameras that track HMDs such as HTC Vive or Oculus Rift worn by the players. The game instance can share this information with other players, playing in different game instances that may be physically separated. However, HMD's tracking data is not enough to mimic the detailed skeletal motion of the subject wearing the HMD.

More realistic avatar based CVEs rely on skeletal tracking. The skeletal tracking can be improved by using multi-camera setups [22]. Using multiple cameras for capturing a subject from different perspectives has become an important research topic in recent years. Several compelling systems for various applications have been proposed for both acquisition and analysis of data from such multi-camera set-ups. Muller et al. [18] proposed a motion capturing system that utilizes six Microsoft Kinect sensors for gait analysis. Their system is based on a client-server model where data from the Kinects is streamed locally over an Ethernet connection to a server machine for visualization and processing. Another system utilizing multiple cameras is FusionKit [22] which is a marker-less skeleton tracking toolkit built using Microsoft Kinect SDK for Microsoft XBox One Kinect sensor. Skeletal information from multiple kinects surrounding a subject is fused to obtain a single skeleton properly aligned with respect to each Kinect. However, none of these systems directly utilize the *point cloud* data to realize a realistic avatar, but they simply transfer the generated skeletons to pre-computed avatar models. Moreover, these systems are not platform independent.

Fusion4D [7], which is a real-time motion capturing pipeline that generates temporally coherent high-quality reconstructions of subjects in a multi-camera setting. Fusion4D eliminates the use of skeletons or template models by reconstructing high-quality meshes from the fusion of point clouds captured from each camera in real-time. It is also, to some extent, independent of the choice of devices used for performance capture, given all devices are of the same type. Fusion4D, although real-time in a local setting, has not been optimized for network transfers which introduces additional challenges like the latency and the network traffic.

An important middleware that inherently addresses networking pipelines for data transfer is the Robot Operating System(ROS). In the ROS ecosystem, components called nodes stream information a.k.a topics to other nodes by using TCP/IP protocol. Other nodes in the system can subscribe to the streamed topics over either a local or a remote network. While this offers a promising solution to stream information from multiple Kinect sensors to remote locations, ROS is more of a generic purpose system that has not been optimized for VR telepresence. Nevertheless, there have been interesting applications of ROS in Virtual Reality, specifically in the field of Human Robot Interaction(HRI), where human motion is applied to robots to allow for their remote operation in environments not accessible to humans. For instance, Inamura et al. [17], successfully utilized cloud platforms to store multimodal, interaction information, which can be fetched to re-create the interaction at a later point in time. A real-time telepresence system leveraging ROS is described in [19]. However, the system is restricted to transferring hand motion by using a single LeapMotion camera.

Sultani et al. [27] streamed point clouds from a single Kinect to an Android client and Linux desktop client. While they did not simultaneously transfer data from multiple Kinect sensors, where the bandwidth requirement is much higher, their system is a good demonstration of transferring point cloud information to remote locations. Figueroa et al. [11] propose InTml (Interaction Techniques Markup Language), a dataflow system for VR applications. Intml

models applications as a series of filters that can be combined by designers, but is not capable of distributed processing.

It is well known that low latency in VR systems and particularly in HMDs is an important factor for presence and for alleviating simulator sickness. Users can develop simulator sickness due to conflicting sensory inputs, for instance, in scenarios where users move in virtual environments while the physical body remains motionless resulting in sensory conflict between visual, somatosensory and vestibular systems [2]. A low-latency system with natural interaction experience that aggregates the components for capturing and transmitting scenes from multiple cameras in an effective manner is therefore crucial for optimum telepresence.

3 OUR DYNCAM LIBRARY

The core of our point cloud-based telepresence system is our DynCam library that processes and distributes data from RGBD sensors. We decided to apply the functional reactive programming (FRP) paradigm for our library. This is appealing because it automatically leads to multithreaded implementations. We start the description of our library with a short FRP recap.

3.1 Functional Reactive Programming

We use FRP to describe the dataflow of the pipeline expressively while simultaneously enabling effortless multithreaded computations. Different implementations of FPR have been used successfully in computer animation, robotics, and GUIs, to name but a few [6, 9, 20, 21]. The idea is that calculations depend on inputs and are automatically triggered when the input data changes. While some implementations deal with continuous and discrete data at the same time, we abstract our data into discrete events. Input events are root nodes in a directed acyclic graph (DAG) and the program's output (.e.g. files, visualization, network, ...) are the leaves. Edges describe the data flow, and inner nodes transform the data. New inputs automatically update all dependent transformations and propagate the result to the output. The main advantage of the FPR paradigm compared to event-based systems that typically use callbacks is the known graph structure of data flow. This knowledge enables an automatic parallelization of tasks that do not depend on each others result.

In our recent DynCam implementation, we use the FRP framework C++React [24]. It supports parallel updates of graph nodes as well as parallel input. C++React distinguishes between two data sources, *signals* and *events*. Values that should stay available over multiple graph traversals are stored in *signals*. If, and only if the value of a signal node changes, all child nodes update their values. Signals are most useful for values that do not change often and are expensive to compute like the extrinsic calibration of a camera. For large, rapidly changing data like images from a camera stream, it is not necessary to explicitly store it in the graph. Instead, we want to quickly hand them to the next node. This behavior is supported by the second type of input nodes, the *events*. *Transformations* take one or more events and signals. When the input of a transformation changes, it updates its result and emits a new event. At the leaf nodes of the DAG, *observers* monitor their inputs. They do not produce any reactive output but display the data on the screen or transmit it over the network.

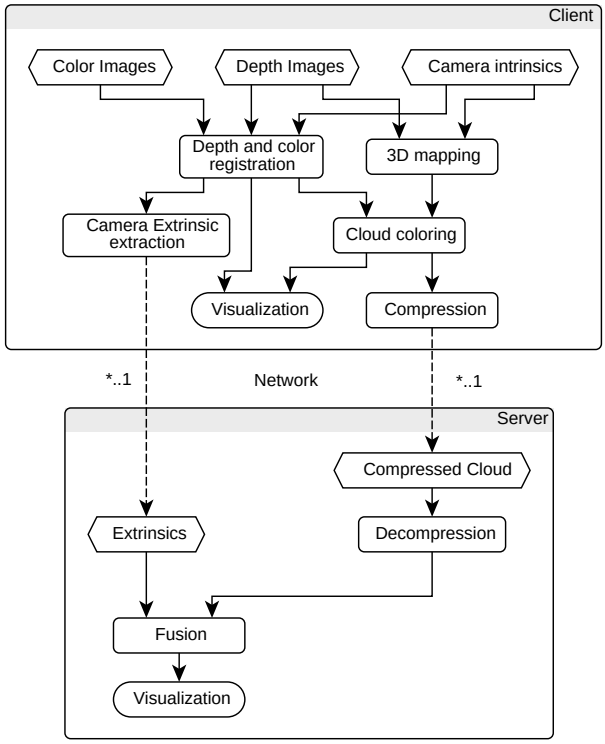

Figure 1: Data flow overview of our pipeline. Data from the RGBD camera are sources on the client. The different transformation stages run concurrently and, thus, update their output as soon as their input changes. Acting as sources, data from multiple clients are then aggregated and displayed on the server.

3.2 DynCam Structure

Unlike most strict linear pipeline architectures, we describe the data flow in our DynCam library following the FRP paradigm. In a telepresence application that is based on RGBD streams, the cameras are the main data sources. Less obvious data sources are the intrinsic and extrinsic camera parameters. These parameters are static and are determined by the depth and color data sources. We distinguish between the dynamic image data streams and the mostly static camera parameters. The image streams are modeled as *events* and the camera parameters as *signals* following the nomenclature of C++React.

The basic structure of DynCam consists of four layers of nodes (see Fig. 1). The root nodes are the interface to the depth camera with a signal for the intrinsic camera parameters and event sources for the depth and color images as described above. From these sources, we compute a 3D point cloud from the depth image and the intrinsic camera parameters. Additionally, we simultaneously register the depth image to the color image. These processes are modeled as *transform* nodes on the event streams in our DAG. This means they are automatically triggered whenever a new frame is generated by events on the root level.

The third level combines the point cloud and registered color image to create a colored point cloud. In another parallel transform node on the level, we use the registered and undistorted color image to search for a calibration pattern to generate the extrinsic camera parameters using the standard procedure of OpenCV.

In the final level, the leaves in our library, we can either send the colored point cloud to the display, or we send it via the network to a remote instance of the library that acts as the server. We decided to send the extrinsic camera parameters and the streaming point clouds independently. The reason for this is that in case of stationary cameras they to not change very often. In this case, we have to send them only once which helps to reduce the network load. However, in case of changing extrinsic parameters, e.g., because of the usage of a Leap motion sensor that is mounted to an HMD to track the user's hands, it is straightforward to send them regularly. In case of external tracking devices, e.g., via Optitrack, we have an option to include this as another signal node in our DAG. In case of network transfer, we additional included the option to compress the point cloud to further reduce the network traffic. We explain the details of our compression method in Section3.3.

In case of a distributed application, an instance of the DynCam library can act as the server that combines the point clouds of all connected DynCam client instances. In this case, the, optionally compressed, points clouds and the respective extrinsic camera parameters are the events and signals, respectively. We have one transform node to decompress the point cloud, and finally, a transform node combines the incoming point cloud to the final shared virtual environment based on the extrinsic parameters.

3.3 Compression

Modern RGBD sensors like the Kinect 2 transmit huge amounts of data. The raw color image (1920 × 1080, 32-bit RGBA) and the depth map (512 × 424, one 32 bit floating value) of the Kinect 2 streamed with 30 frames per second produce 2.048 Gbit of data per second. Even the reconstructed colored point cloud, i.e., 512 × 424 3D points with float accuracy and 24-bit RGB color would result in 745 Mbps which would almost completely occupy a traditional 1 Gbit ethernet connection. Consequently, reduction of the data is necessary to enable a multi-camera streaming application in normal networks. A usual method to reduce the amount of data is the application of compression algorithms. In our real-time environment, the compression and decompression speed of the algorithms is an essential requirement.

We decided to use different compression algorithms for the RGB data and the point cloud. For the point cloud data, we use a lossy compression with a negligible loss of accuracy: The maximum range of the Kinect 2 sensor is limited to 4.5 m [16], and the accuracy at 3 meters distance is in the range of −8 mm to 37 mm [23]. We, therefore, quantize the 3D position components to 1 mm. This reduces half of the space required per point from 3 × 32bit floats to 3 × 16bit.

For the RGB point colors, we use JPEG to compress the color image that was mapped onto the depth map. While this lossy compression reduces the quality, it is fast enough for real-time application. However, we can simply switch to lossless PNG or WEBP compression if required.

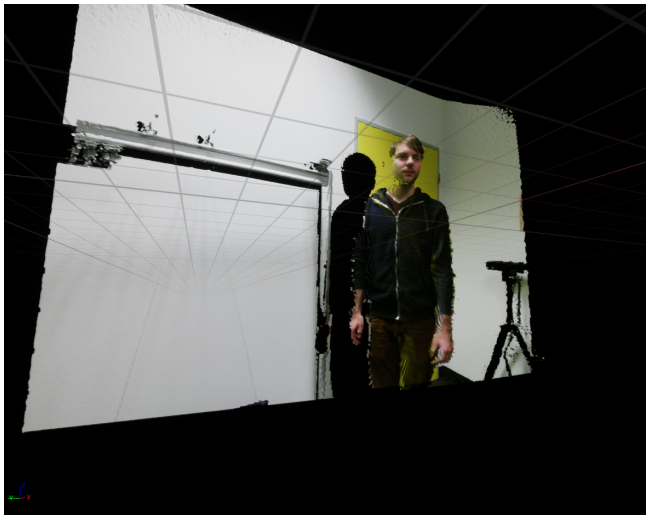

Figure 2: Our DynCam library is available as Unreal Engine 4 plugin and can render dynamic point clouds from a local RGBD camera, or a cloud streamed over the network.

In addition to the individual compression methods mentioned above, we use LZ4 [5] for additional compression after quantization. LZ4 is a lossless compression algorithm with focus on fast compression and has been evaluated for UHD 3D video transmission before [13].

4 USE CASE: MULTI-USER VR-PLUGIN FOR UNREAL 4

Our library is lightweight and easy to integrate into existing applications. As an example, we have implemented a plugin for the recent Unreal Engine 4 (UE4) [10]. This allows to either directly connect an RGBD camera to the PC and visualize the point cloud in UE4 or to stream and visualize clouds transmitted via the network. We will start with a short description of our integration to Unreal before we describe our visualization of the point cloud.

4.1 Integration

We have integrated DynCam into UE4 using the UE4 plugin system. The access point to the DynCam interface is straightforward. During initialization, we have to decide whether we want to use DynCam as server or client. For a locally connected RGBD camer,a we instantiate a new camera and pass it to a point cloud mapper. When the point cloud data should be sent over the network, we create a server instance. During runtime, we can access an event stream that provides point clouds as a simple vector every time a new cloud is available. This stream is monitored by an observer who takes a lambda or function pointer as an argument and calls the function every time new data is available. In our UE4 plugin, we cache the last point cloud from the lambda. This guarantees that DynCam does not interfere with ongoing rendering operations.

4.2 Visualization

UE4 does not natively support rendering of point clouds. Surprisingly, it turned out to be relatively complicated to implement it. In this section, we want to share our experience with the research community. Unreal has an integrated visualization of particle systems. This can consist of millions of animated particles which are rendered in real-time. It seems to be obvious to use this also for our point clouds. The Unreal particle renderer gains its efficiency from a pure GPU implementation, i.e., the position updates of each particle are solely computed on the GPU; hence it can be hardly applied to our point clouds transferred over the network. Another option we explored was to utilize instancing. Basic tests of 512×424 instanced quads with a simple animation already caused the frame rate to drop to less than 30 frames per second. Another option we considered was to use a vertex buffer with three vertices per point, forming a triangle around the point and update it consecutively. This method turned out to be slightly faster than the instancing method, but the rendering speed was still much slower than 45 frames per second which would be the minimum for HMD-based VR applications.

Finally, dynamic textures and shader-based splatting turned out to be the most efficient method. The basic idea is to use two different textures, one for the color information and one the 3D position. The textures are updated whenever a new point cloud arrives. UE4 allows only textures of square size. Consequently, we map the 1D array of colors and positions generated by DynCam to the 2D textures. While for the colors a four channel texture with 8bits is sufficient, we use a 32bit texture per channel for the positions.

To initialize the textures, we generate a static vertex buffer on the GPU that contains N triangles where N is the number of points in our point cloud. During runtime, we identify the triangle's id $i \in N$ by its z position and use this as the index to look up the corresponding position and color in the texture. This can be easily done in a vertex shader. The triangles' vertices $V_j, j \in 3N$, are centered around the xy-origin, and their z-position corresponds to the vertex number. In the shader, we can then identify the triangle id i as follows:

$$i(v) = v.z \; div \; 3 \tag{1}$$

where v is the current vertex. With the index, we read the triangle's position from the position texture and offset the vertex accordingly. The color is determined in the same way. To generate circular splats instead of triangles, we simply discard all pixels from the triangle that are not part of the triangle's incircle in the pixel shader. Finally, we have to discard triangles that are not used because the number of points in the point cloud is smaller than the size of the texture. However, this can be easily solved by using a uniform to store N and allows the use of a static vertex buffer.

This method has a small drawback: UE4's maximum texture size is currently restricted to 8k $\times$ 8k pixels, thus limiting us to $\approx 67k$ points per cloud. However, this seems to be sufficient even for large scenes.

5 EVALUATION

We have implemented our DynCam library using C++. Our implementation is platform independent, and we tested our library on several current Microsoft Windows and Linux systems. In this section, we will present some performance measurements. The most influential measures are the latency and the network traffic produced by DynCam. While there exist many tools to track the network traffic, latency measurements are more complicated. We decided to apply a method described in [25]. However, we extended this method in several points which makes it much more robust. We ported the Matlab code to Python, use OpenCV's blob detection and better initialize the optimization step. Our code is available here[1]. We will shortly sketch this in the next subsection because this could be also interesting for other researchers that want to perform latency measurements before we finally present our experiments and the results.

Figure 3: Latency measurement with a LED mounted on a pendulum. The Kinect 2's output is displayed on the screen. Both the LED on the pendulum and the recorded LED are filmed by an ordinary camera.

Table 1: End to end latency from Kinect 2 to screen and required bandwidth of the tested configurations. Entries marked with 2D did not reconstruct the point cloud but only displayed the color stream from the Kinect. In the last three measurements, the Kinect data is transferred over gigabit ethernet.

Configuration	Mean latency ms		Required bandwidth Mbps
Protonect 2D	87	± 10	
DynCam 2D	101	± 4	
MSKinect 2D	134	± 9	
DynCam	124	± 11	
ROS	136	± 6	
LAN			
DynCam uncompressed	146	± 8	773
DynCam compressed	136	± 11	314
ROS uncompressed	215	± 7	901

[1] http://cgvr.cs.uni-bremen.de/papers/vric2018

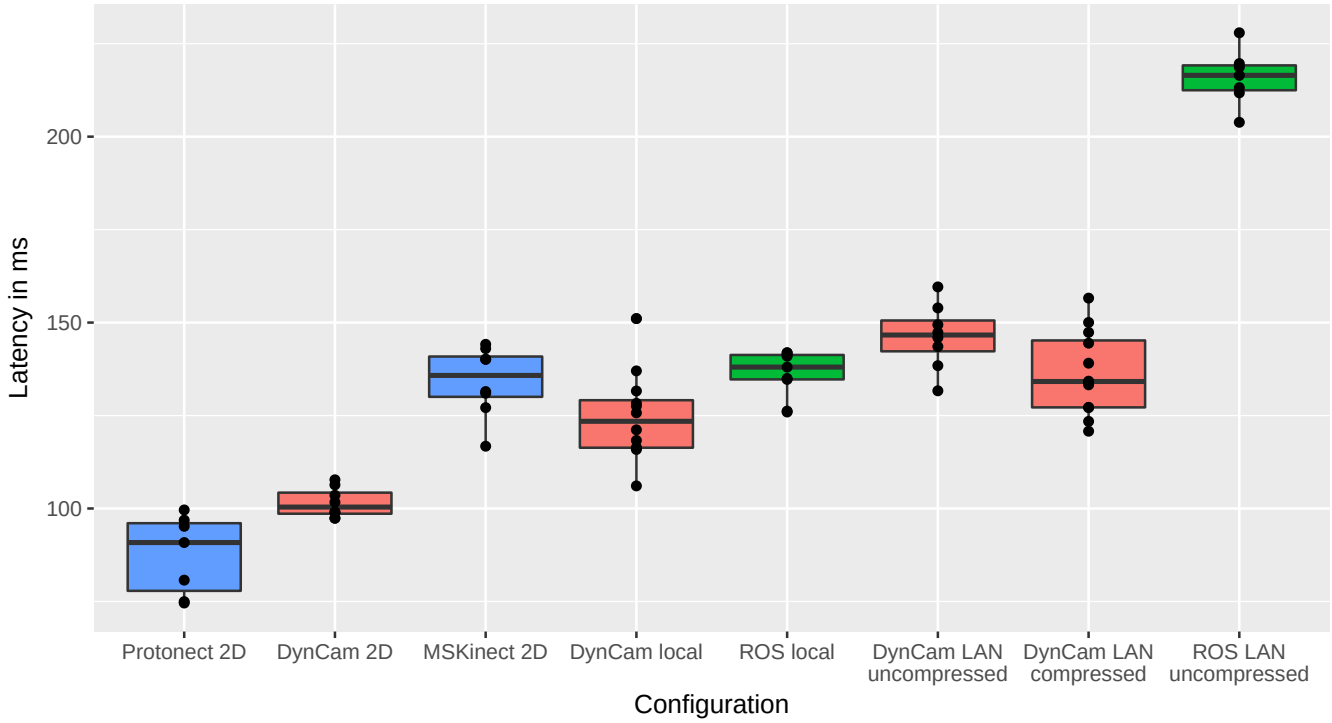

Figure 4: End to end latency from Kinect 2 to screen of the tested configurations. Entries marked with 2D did not reconstruct the point cloud but only displayed the color stream from the Kinect. In the last three measurements, the Kinect data is transferred over gigabit ethernet. Blue entries are the samples provided by the open and closed source Kinect drivers. Red entries denote out results and green are measurements of ROS.

5.1 Improved Latency Measurement

The basic idea of our end to end latency measurements was proposed by Steed [25]: We set a simple LED light that is attached to a pendulum into motion. Then we record the motion with a Kinect 2. Simultaneously, we display the live video stream of the Kinect to a screen that is placed next to the pendulum. A regular camera records the pendulum and the visualization on the screen (see Figure 3). The recording of this camera is analyzed using computer vision techniques to measure the latency between the pendulum and the screen. Obviously, we can simply display the Kinect stream on the same PC or route through a network to measure the network latency.

Unfortunately, the original Matlab scripts provided by Steed [26] are not compatible with the most recent Matlab 2016 version. We tried to adapt it to the current packages but the localization of the LED was not reliable, and the sinus fitting failed due to the noise. Consequently, we reimplemented the algorithm to extend it by a more robust filtering and better initialization of the optimization. The basic idea remains the same. In every frame of that recorded video, we detect the position of the two LEDs. By using a pendulum, we can model the movement by a sinusoidal of the form:

$$y(x) = A\sin(2\pi f x + \Phi) \tag{2}$$

The deflection y at frame x depends on the frequency of f and the phase angle Φ. In order to keep $A = 1$ constant, we normalize the range of the pendulum. In contrast to Steed's approach, our formulation keeps the phase angle between 0 and 2π. This allows us to limit our optimization independent of the video's frame rate. Moreover, we filter the tracked marker positions with a moving median and a window of 5 frames (i.e., 100ms in our recordings). Additionally, we propose the following automatic initialization of the parameters to reduce the failure rate of the optimization. For the initialization, we transform the signal into Fourier space and extract the dominating frequency as well as the corresponding phase shift. Finally, we can calculate most likely frequency and phase angle by minimizing the difference between the recorded movement and the sinus model.

In the video, one marker is ahead of the other by a time shift. As the frequencies of both markers are the same, the difference in the phase angles $\Delta\Phi$ corresponds to the latency of the system. Depending on the frequency and the frames per second (r) of the video, we calculate the final latency Δt in seconds by:

$$\Delta t = \frac{\Delta\Phi}{2\pi f r} \tag{3}$$

In a more recent publication, Fristopn and Steed [12] improve the original method by correlating the signals instead of fitting a sine function. In our experiments, this proved to be less reliable due to the noisy signal from the point clouds, however.

5.2 Results

We have measured the latency of our library under different configurations and compared it to the sample applications of the closed and open source Kinect drivers in the single PC case. Furthermore, we have compared our library to the performance, i.e., the latency as well as the network traffic, of ROS. ROS is a popular middleware for distributed environments. The ROS system is comprised of processes called nodes that are capable of running on any machine in the distributed ROS system. Nodes communicate with each other by publishing data as topics or by subscribing to topics that were already published by other nodes. By design, this communication takes place using TCP based networks, similar to our network implementation. We used the IAI Kinect2 package [31] for the Indigo release and rosviz for the visualization (see Figure Fig. 5). We chose ROS as it is widely used for point cloud processing in robotics and is available for Linux. Measurements for the official Kinect SDK were taken on Windows 10. All other evaluations were taken on Arch Linux. The PC was equipped with an Intel i7-7800X CPU and a Nvidia GeForce GTX 1080 TI. The measured latencies are listed in Table 1 and compared Fig. 4. In the following will describe the results of our measurements in detail.

In a first, we have measured the end to end latency that is caused by a single Kinect 2 and the display connected to the same PC. In this configuration, we test the open source libfreenect2 driver [4] using the *Protonect* application that comes with it as well as the official Microsoft driver [15] (*MSKinect*). In both cases, we only measured the latency for the color stream because Protonect does not show the point cloud. We have compared this baseline to our DynCam which internally uses libfreenect2 to connect to the Kinect. The results show that the open source driver libfreenect2 has much lower latency (87ms) compared to the original Kinect driver (134ms). DynCam uses the same driver as Protonect but does more processing, which results in 14ms higher latency. In all cases, except for Protonect, the latency is larger than 100ms not considering the depth image. This could be problematic in VR applications.

In our second experiment, we generated a colored point cloud and displayed it on the same PC where the Kinect was connected to. This adds additional 13ms of latency for DynCam. However, it is still faster than ROS (136ms). This significantly aggravates when measuring the network performance: In this scenario, we connected the Kinect to a PC that was connected to the display PC via the local network with 1 GBit capacity. In this case, we gain a latency of DynCam that is 69ms smaller than that of ROS. Turning on compression reduces the latency gain of DynCam and makes it as fast as ROS without network. Even though more processing power is required for compression end decompression, the reduced traffic and parallel processing reduce the latency. Even the uncompressed DynCam network traffic is smaller than that of ROS (SI773Mbps vs. 901 Mbps. However, in both cases, the network operates at almost full capacity which results in only a single point cloud stream that can be transferred through the 1 GBit connection. The compressed DynCam stream only requires 314 Mbps. This allows connecting multiple DynCam instances to the same network.

Figure 5: Setup for the distributed latency measurement.

6 CONCLUSIONS AND FUTURE WORKS

We have presented DynCam, a new lightweight, platform independent library for real-time, low latency, streaming point cloud processing. Our library gains its efficiency from the functional reactive programming paradigm that automatically enables multithreading of different data transformations within the processing pipeline. Our library supports the generation of fused point clouds from multiple RGBD sources. These streaming images can be even gathered via network connection thanks to an efficient compression algorithm. Our results show that the latency of DynCam is lower than that of the original Microsoft Kinect application when used on the same PC and also lower than the distributed system ROS when used with an ethernet connection.

In order to measure the latency, we have presented an improvement of the current stat-of-the-art method for latency measurements which reduces the parameter tuning and is more robust. Finally, we have presented a plugin of DynCam for the Unreal Engine 4. To our knowledge, this is also the first real-time visualization of point clouds sourced either from a local depth sensor or transmitted via ethernet. The Unreal plugin as well as the python code for the latency measurement are available on our homepage[2].

However, our work also opens interesting avenues for future research. For instance, we plan to enhance the point cloud rendering in Unreal. The current approach is fast but does not support advanced rendering techniques like shadows, global illumination or ambient occlusion. Moreover, it will be especially interesting to study the impact of point cloud rendering techniques in VR on the rendering performance and perceived quality. Also, the user perception of point cloud avatars would be an interesting topic for research. Moreover, we want to implement segmentation on the clients to reduce the bandwidth requirements and allow the use of specialized fusion algorithms.

REFERENCES

[1] Jascha Achenbach, Thomas Waltemate, Marc Erich Latoschik, and Mario Botsch. 2017. Fast generation of realistic virtual humans. In *Proc. 23rd ACM Symp. Virtual Real. Softw. Technol. - VRST '17*. ACM Press, New York, New York, USA, 1–10. https://doi.org/10.1145/3139131.3139154

[2] Cassandra N. Aldaba, Paul J. White, Ahmad Byagowi, and Zahra Moussavi. 2017. Virtual reality body motion induced navigational controllers and their effects on simulator sickness and pathfinding. In *2017 39th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.* IEEE, 4175–4178. https://doi.org/10.1109/EMBC.2017.8037776

[3] Jovis Joseph Aloor, P S Sahana, S Seethal, Sneha Thomas, and M.T Rajappan Pillai. 2016. Design of VR headset using augmented reality. In *2016 Int. Conf.*

[2] http://cgvr.cs.uni-bremen.de/papers/vric2018

Electr. Electron. Optim. Tech. IEEE, 3540–3544. https://doi.org/10.1109/ICEEOT.2016.7755363

[4] Joshua Blake, Lingzhu Xiang, Florian Echtler, and Christian Kerl. 2015. libfreenect2: Open source drivers for the Kinect for Windows v2 device. (2015).

[5] Yann Collet. 2011. Lz4. (2011). Retrieved December 17, 2017 from https://github.com/lz4/lz4

[6] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implement. - PLDI '13*. ACM Press, New York, New York, USA, 411. https://doi.org/10.1145/2491956.2462161

[7] Mingsong Dou, Jonathan Taylor, Pushmeet Kohli, Vladimir Tankovich, Shahram Izadi, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, and David Kim. 2016. Fusion4D. *ACM Trans. Graph.* 35, 4 (jul 2016), 1–13. https://doi.org/10.1145/2897824.2925969

[8] Riley Dutton. 2017. OrbusVR :: Fantasy virtual reality MMO for HTC Vive and Oculus Rift + Touch. (2017). Retrieved December 17, 2017 from https://orbusvr.com/

[9] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proc. Second ACM SIGPLAN Int. Conf. Funct. Program. - ICFP '97*. ACM Press, New York, New York, USA, 263–273. https://doi.org/10.1145/258948.258973

[10] Epic Games. 2014. Unreal Engine 4. (2014). Retrieved December 17, 2017 from https://www.unrealengine.com

[11] Pablo Figueroa, Walter F Bischof, Pierre Boulanger, H James Hoover, and Robyn Taylor. 2008. Intml: A dataflow oriented development system for virtual reality applications. *Presence: Teleoperators and Virtual Environments* 17, 5 (2008), 492–511.

[12] Sebastian Friston and Anthony Steed. 2014. Measuring latency in virtual environments. *IEEE transactions on visualization and computer graphics* 20, 4 (2014), 616–625.

[13] Ruan Delgado Gomes, Yuri Gonzaga Gonçalves da Costa, Lucenildo Lins Aquino Júnior, Manoel Gomes da Silva Neto, Alexandre Nóbrega Duarte, and Guido Lemos de Souza Filho. 2013. A solution for transmitting and displaying UHD 3D raw videos using lossless compression. In *Proc. 19th Brazilian Symp. Multimed. web - WebMedia '13*. ACM Press, New York, New York, USA, 173–176. https://doi.org/10.1145/2526188.2526228

[14] Alexander Kulik, André Kunert, Stephan Beck, Roman Reichel, Roland Blach, Armin Zink, and Bernd Froehlich. 2011. C1x6: a stereoscopic six-user display for co-located collaboration in shared virtual environments. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 188.

[15] Microsoft. 2012. Kinect Studio. (2012). Retrieved December 17, 2017 from https://msdn.microsoft.com/en-us/library/hh855389.aspx

[16] Microsoft. 2014. Kinect hardware. (2014). Retrieved December 17, 2017 from https://developer.microsoft.com/en-us/windows/kinect/hardware

[17] Yoshiaki Mizuchi and Tetsunari Inamura. 2017. Cloud-based multimodal human-robot interaction simulator utilizing ROS and unity frameworks. In *System Integration (SII), 2017 IEEE/SICE International Symposium on*. IEEE, 948–955.

[18] Björn Müller, Winfried Ilg, Martin A Giese, and Nicolas Ludolph. 2017. Improved Kinect sensor based motion capturing system for gait assessment. *bioRxiv* (2017), 098863. https://doi.org/10.1101/098863

[19] Lorenzo Peppoloni, Filippo Brizzi, Carlo Alberto Avizzano, and Emanuele Ruffaldi. 2015. Immersive ros-integrated framework for robot teleoperation. In *3D User Interfaces (3DUI), 2015 IEEE Symposium on*. IEEE, 177–178.

[20] J. Peterson, G.D. Hager, and P. Hudak. 1999. A language for declarative robotic programming. *Proc. 1999 IEEE Int. Conf. Robot. Autom. (Cat. No.99CH36288C)* 2 (1999), 1144–1151. https://doi.org/10.1109/ROBOT.1999.772516

[21] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. 1999. Prototyping real-time vision systems. In *Proc. 21st Int. Conf. Softw. Eng. - ICSE '99*. ACM Press, New York, New York, USA, 484–493. https://doi.org/10.1145/302405.302681

[22] Michael Rietzler, Florian Geiselhart, Janek Thomas, and Enrico Rukzio. 2016. FusionKit. In *Proc. 8th ACM SIGCHI Symp. Eng. Interact. Comput. Syst. - EICS '16*. ACM Press, New York, New York, USA, 73–84. https://doi.org/10.1145/2933242.2933263

[23] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. 2015. Kinect range sensing: Structured-light versus Time-of-Flight Kinect. *Comput. Vis. Image Underst.* 139 (2015), 1–20. https://doi.org/10.1016/j.cviu.2015.05.006 arXiv:arXiv:1505.05459v1

[24] Sebastian Schlangster. 2015. C++React. (2015). Retrieved December 15, 2017 from https://github.com/schlangster/cpp.react/tree/legacy1

[25] Anthony Steed. 2008. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proc. 2008 ACM Symp. Virtual Real. Softw. Technol. - VRST '08*. ACM Press, New York, New York, USA, 123. https://doi.org/10.1145/1450579.1450606

[26] Anthony Steed. 2008. Latency Measurement Demonstration. (2008). Retrieved December 8, 2017 from https://wp.cs.ucl.ac.uk/anthonysteed/research/tech/latencydemo/

[27] Zainab Namh Sultani and Rana Fareed Ghani. 2015. Kinect 3D Point Cloud Live Video Streaming. In *Procedia Comput. Sci.*, Vol. 65. Elsevier, 125–132. https://doi.org/10.1016/j.procs.2015.09.090

[28] Subarna Tripathi and Brian Guenter. 2017. A Statistical Approach to Continuous Self-Calibrating Eye Gaze Tracking for Head-Mounted Virtual Reality Systems. In *2017 IEEE Winter Conf. Appl. Comput. Vis.* IEEE, 862–870. https://doi.org/10.1109/WACV.2017.101

[29] Sarah F. van der Land, Alexander P. Schouten, Frans Feldberg, Marleen Huysman, and Bart van den Hooff. 2015. Does Avatar Appearance Matter? How Team Visual Similarity and Member-Avatar Similarity Influence Virtual Team Performance. *Hum. Commun. Res.* 41, 1 (jan 2015), 128–153. https://doi.org/10.1111/hcre.12044

[30] Thomas Waltemate, Dominik Gall, Daniel Roth, Mario Botsch, and Marc Erich Latoschik. 2018. The Impact of Avatar Personalization and Immersion on Virtual Body Ownership, Presence, and Emotional Response. *IEEE Transactions on Visualization and Computer Graphics* (2018).

[31] Thiemo Wiedemeyer. 2015. IAI Kinect2. (2015). Retrieved November 11, 2017 from https://github.com/code-iai/iai