IBC2022

DATA COMPRESSION FOR 6 DEGREES OF FREEDOM VIRTUAL REALITY APPLICATIONS

S. Poularakis, K. Mockford and G Meardi

V-Nova Limited, UK

ABSTRACT

6 Degrees of Freedom (DoF) are used in Virtual Reality (VR) applications to enhance the user experience compared to the standard 3DoF solutions. Due to its sparse nature, 6DoF information is typically represented in a point cloud form, where each element describes the position of a point in the 3D space, as well as its attributes (e.g., colour and transparency). Although it enhances user experience, 6 DoF requires a higher volume of data compared to 3DoF, which has made content distribution challenging and has also limited its applications to high-end specialised machines.

The aim of our work was to design a novel point cloud compression scheme to allow 6DoF VR applications to run in real-time on high-end consumer devices, such as gaming laptop and desktop machines. Although our solution was designed specifically for the PresenZ 6 DOF VR movies format, it may be easily applied on other volumetric video formats as well.

INTRODUCTION

In a typical Virtual Reality (VR) scenario, Degrees of Freedom (DoF) are used to track the motion of a headset-wearing user within a three-dimensional (3D) space and adjust accordingly the image that the user views. 3 DoF applications track only rotational movement around the x, y, and z axes (known as pitch, yaw, roll), while 6DoF applications also track translational movement (surging, swaying, heaving), allowing for additional effects, such as moving forward/backward, left/right, and up/down [1, 2]. In addition to enhanced user experience, 6DoF VR can help reduce motion sickness and feelings of disorientation, by providing a better sense of presence [10].

Due to its sparse nature, 6 DoF information is typically represented in a point cloud form, where each element describes the 3D position of a point, as well as its color, transparency, orientation, and motion. It may also contain additional data, such as information about the camera(s) used to capture the 3D view. The actual number of

points depends on the complexity of the visual scene: a typical frame may consist of over 5 million points.

Although it enhances user experience, 6 DoF requires a higher volume of data compared to 3DoF, which has made content distribution challenging and has also limited its applications to high-end specialised machines. The key challenges that one needs to address are: 1) high data entropy, which typically exceeds the capacity of conventional communication channels, such as the 500 MB/s of Solid-State Drives (SSD) [3], and 2) real-time video rendering requirements at relatively high frame rates (30 fps). In this work, we describe our approach towards addressing the above challenges using a novel data compression scheme, designed specifically for point cloud datasets.

Our data compression format describes each frame individually, and consists of a fixed header layer, as well as several optional data layers. The fixed header layer describes basic information, such as the number of points and the used color space, as well as the types of coding tools and techniques used for various point cloud subgroups and their attributes. Depending on the information included in the fixed header, additional header layers may be present in the bitstream, further describing encoding methods, parameters, and metadata. Finally, additional core layers are used to store the encoded values for each attribute.

We also designed and implemented a codec API, that allows encoding of a series of point cloud frames and decoding it in real-time on high-end laptops and gaming desktop machines. Our actual encoder and decoder implementations were developed in C++, utilizing techniques such as multi-threading and IntelTM Single-Instruction-Multiple-Data (SIMD) intrinsics [4, 5].

This paper discusses background work for point cloud compression and VR applications, then describes our approach in detail, our experimental results, conclusions and discusses potential further developments.

BACKGROUND AND RELATED WORK

Presenz VR system

PresenZ [6] is a volumetric movie format that allows 6 degrees of freedom in precomputed images with a Virtual Reality (VR) headset. Unlike 360° movies, with PresenZ the viewer is able to move inside the image and get closer to objects or characters, creating a real sense of scale and immersion, feeling like you are part of the action.

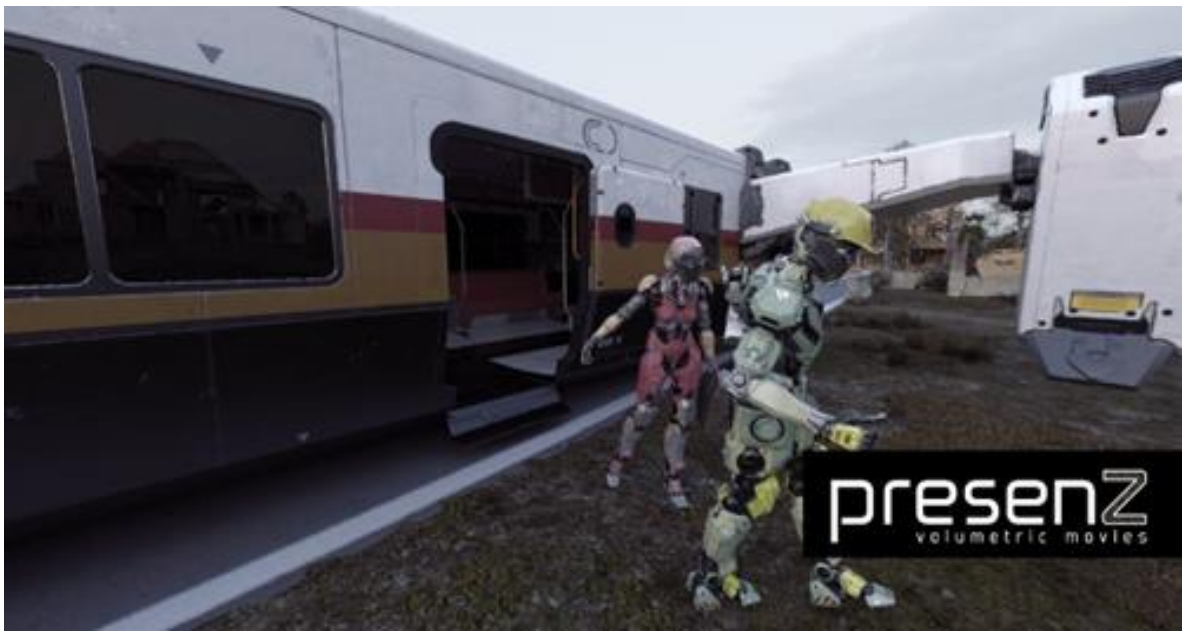

Figure 1- A screenshot from the "Construct" VR movie (courtesy of Presenz)

PresenZ VR technology enables the creation of VR images and movies with 6 DOF from conventional 3D models via point cloud datasets. The created point clouds are then rendered in real time and displayed in a VR headset allowing viewing in 6DoF. The point cloud framerate is usually 25 or 30 fps, and the points that are moving in each frame contain motion vectors to interpolate between frames to allow smooth movement in VR.

An essential element of PresenZ VR is the Zone of View (ZOV). The ZOV is a volume box, inside which the VR user can move freely and have a proper perspective change on the pre-rendered 3D scene. PresenZ makes a render from the ZOV, gathering all the information that is possible to see from within this “zone”. In contrast, a 3DoF VR system uses a perspective camera projection centred around a single point of view (or focal point), limiting the user’s experience, even if the VR headset has positional tracking. This is the main cause of cybersickness, discomfort and poor immersion in 3DoF VR systems [6].

The PresenZ VR system originally used the BLOSC [8] algorithm to compress the 6DoF point cloud data. While BLOSC offers fast decoding, it results in high bitrates (typically higher than 500 MB/s), which limits this approach to expensive systems equipped with NVMe SSD drives.

Description of Presenz VR data fields

The position of each point in a point cloud is described by its X, Y, Z coordinates in a Cartesian space. The X and Y coordinates correspond to a 2D projection on a sensor (such as a camera), while the Z coordinate relates to the distance from the camera. Each point has multiple data attributes, which include color information for the left and right eye, the size and opacity of the point, a normal vector [11], a camera index, and motion vectors to describe point motion in time. Typically, most of the points have the same value for both eyes, although values may differ such as when there are stereo reflection effects on window and other reflective surfaces.

A VR movie is formed as a sequence of consecutive point cloud sets (termed as frames). The number of points in each frame varies depending on how many points will be covered (occluded) and uncovered by camera movements. For example, when there are thin objects close to the viewer, more objects can be uncovered as the camera is allowed to move upwards/downwards/left/right/forward/backwards and capture points behind the close object. Most frames contain around 4 to 5 million points.

Some attributes, such as the left and right eye colour, can be encoded in a lossy manner without a significant degradation of the VR user experience and the perceived visual quality. On the contrary, some other attributes, such as size and camera index, must remain intact, otherwise very noticeable errors will appear when viewing the point cloud (e.g. points being in the wrong position or holes appearing in objects).

Presenz VR Point Cloud Codec Requirements

To enable a 6DOF VR movie to be played on a high-end gaming PC or laptop equipped with a conventional SSD drive requires the compressed VR movie to have a data rate lower than the 500 MB/s whilst keeping the decoding processes simple enough to allow 30 frames per second to be decoded, and for the video quality to be excellent to provide a truly immersive experience. Hence the need for a data compression codec tuned for point clouds.

Point cloud compression

A point cloud is a flexible way to represent a set of individual 3D points. Each point has a 3D position as well as some other attributes such as colour, surface normal, etc [9].

Due to the wide range of point cloud data-based applications, the Moving Picture Experts Group (MPEG) began developing 3D point cloud compression standards in 2017. MPEG's work is divided into two parts [9, 14]: Video-based point cloud compression (V-PCC), which is appropriate for point sets with a relatively uniform distribution of points, and Geometry-based (G-PCC), which is appropriate for more sparse distributions. G-PCC is the most relevant to our work, as we also focus on sparser distributions.

MPEG G-PCC [7, 18] provides several alternative methods for to signal the occupied points in the point cloud are occupied including a standard octree representation, and a direct (x, y, z) representation, and a combination of octree and a triangular surface representation known as triangle soup.

MPEG G-PCC also defines several methods for attribute encoding, including Region Adaptive Hierarchical Transform (RAHT) which is similar to wavelet transforms in the 2D

space, and Layer of Detail (LoD) generation where an attribute is predicted from already coded points in the current or subsequent layers.

Another codec that we considered was Google Draco [15], an open-source mesh and point cloud compression codec. The techniques for encoding point cloud occupancy are similar to G-PCC except that kd-tree structures are used rather than octrees. Draco uses similar techniques to G-PCC for coding the attributes.

In the spring of 2020, when work started on developing a point cloud compression codec to meet Presenz VR's requirements, MPEG G-PCC was still at an early stage of development, and so was judged to not have all the tools and techniques necessary to fulfil the requirements. One of the key missing capabilities was a temporal frame prediction scheme, although it is a planned G-PCC feature [18]. Google Draco also does not include a temporal prediction mode.

Hence the decision was made to develop a proprietary solution, utilizing the in-house knowledge gained through the development of hierarchical codecs such as SMPTE ST-2117 VC-6 [16] and MPEG-5 Part 2 (LCEVC) [17].

Similar to G-PCC, our scheme uses either octrees or direct (x, y, z) representations for coding the geometry of the occupied points in the point cloud, and various prediction and arithmetic encoding methods to encode the point attributes.

Our scheme organizes the data into independent subsets and the techniques used are designed to allow massive parallelization of the coding process. It also supports both intra and temporal prediction, allowing the codec to take advantage of point similarities across consecutive frames and therefore encode points at much lower bitrates compared to the intra mode. Additionally, we wanted our scheme to be flexible, so that it can be easily extended to encode different point cloud formats.

OUR APPROACH

General structure

Our approach begins by encoding the position of each 3D point within each point cloud frame. This encoding reorders the points, with the new order being preserved throughout the encoding and decoding processes. The purpose of the point reordering is to take advantage of any structural similarities between adjacent points and thus achieve better compression through differential coding and other similar techniques. Each attribute is then encoded separately.

When there is a sequence of frames, the first frame is encoded as above (termed as Intra mode). Each subsequent frame may be encoded in Intra mode too, or it may be encoded based on its preceding frame, where points can be copied between frames with a correction for the colour, providing improved compression and decoding speed. We term the later mode as Temporal mode.

Compression methods

Octree representation

An octree may be used to encode the X, Y, and Z coordinates of each point. Starting with a cube (with sides with a length of a power of two that is large enough to enclose every point), the points are separated into 8 equally sized sub-cubes (the length of the sub-cube sides will be half the length of the sides of the original cube). 8 bits are used to show which of these 8 sub-cubes contain points, using a 0 to indicate that no points are in the sub-cube and a 1 to show there is at least one point included. The order of the bits, starting from the least significant one, is given by the order that the cubes are checked. The first cube checked is the one closest to (0,0,0). This splitting is repeated, splitting each non-empty cube to 8 equally sized cubes, until the side length of the cubes becomes a predetermined power of two.

Each of the cubes in the final layer of the octree is referenced by its top left corner coordinates. The 8-bit codes show how the octree was split and are entropy encoded. The order of the codes is given in a Breadth-First-Search (BFS) manner for the first seven levels of the octree. Finally, for each actual data point, we encode its difference to the top-left corner coordinates of the smallest containing cube.

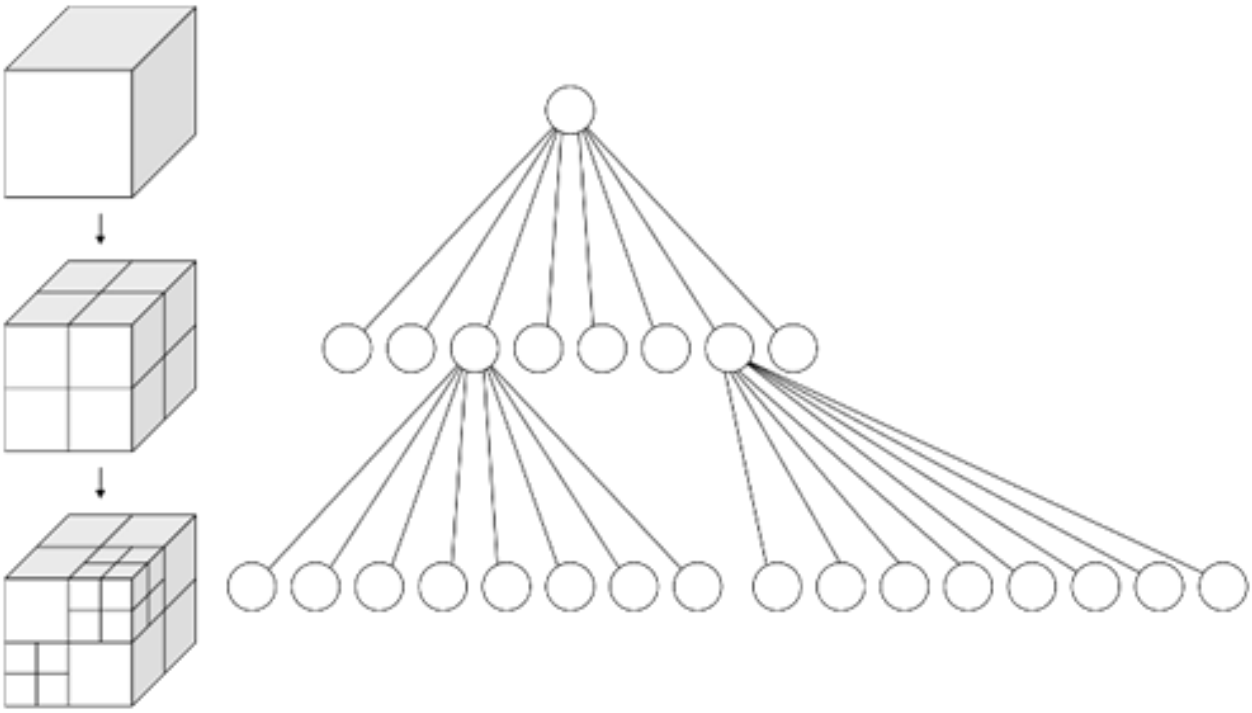

Figure 2 - An illustration of an octree and how it splits a 3D cube into sub-cubes [12]

XYZ coding

If an octree is not used for encoding the position of each point, then the X, Y, and Z values can be encoded separately. The first step is sorting the point coordinates and splitting them to high (most-significant) and low (least-significant) bits. The actual number of low bits is defined in the header (typically 2 or 4). When ignoring the low bits, any duplicates in the

combined high bits of X, Y, and Z are counted and removed. The high bits then give the positions of the cubes that would be found at a corresponding level of an octree. They may then be differentially coded (which is signalled in the bitstream) and finally entropy encoded. The low bits are handled in the same manner as in the octree method where they are given a fixed number of bits per value, and they are added directly to the bitstream.

Pre-processing

After the values of each attribute have been sorted according to the order given by the position encoding method, the values can be pre-processed to facilitate later encoding. The possible methods used in this step are: (1) differential coding, and (2) grouping the points (normally based on the octree) and then subtracting an average from each group of values of the attribute. These average values for each group are then differentially coded and entropy coded. The attributes that can be lossy encoded may be quantized at some user-defined or data-adaptive levels.

Sparsification

After the pre-processing of the attribute values, it is likely that there will be many zeros. To reduce entropy and the number of values that need to be decoded, we introduce a method termed as sparsification. The sparsification method groups multiple values together, checks if a whole group is zero-valued, and if so then this zero group can be removed from further encoding.

For a fixed size N, each data subgroup of size N, starting from the beginning and moving along N values each time, is checked to see if all the values are zero. A bitarray is created, containing a 1 for each group that contains any non-zero values, and a 0 for each group that contains only zeros. This bitarray is subsequently run-length encoded, where only the first symbol, which is either a 0 or 1, and the length of each run are needed. There are only two possible values in the bitarray and so the value of each run alternates between these.

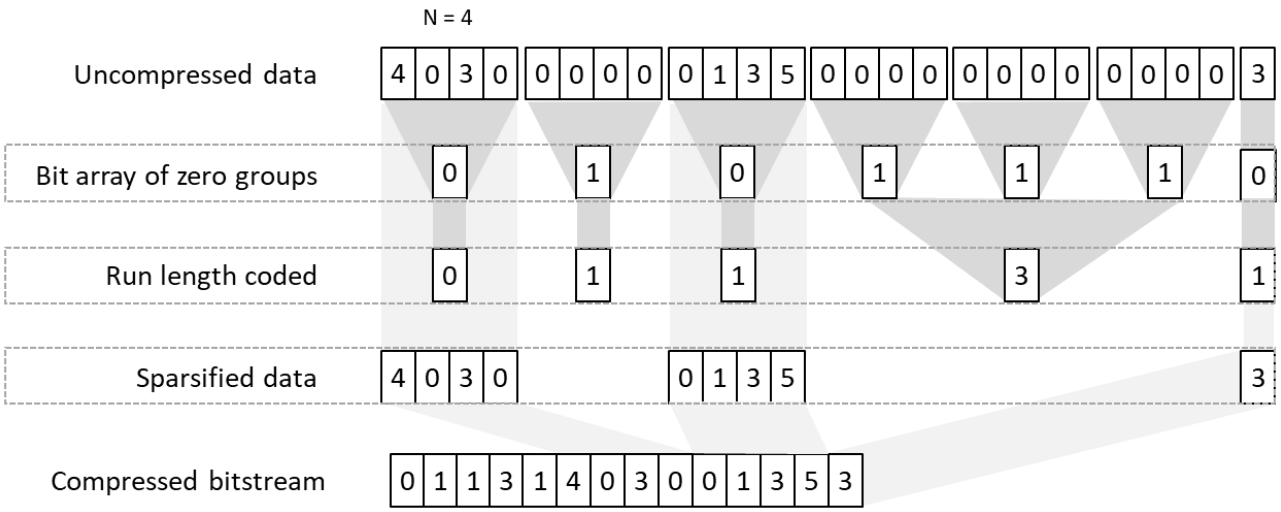

Figure 3 An example of sparsification with N = 4

Figure 3 shows an example with N = 4 and how the bitarray that identifies the groups containing all zeros is run length encoded.

Entropy coding
After pre-processing and sparsification, entropy coding methods are used to encode the data of each attribute, as well as information about the octree and temporal scheme. The entropy coding method used is either Huffman or range encoding. While range encoding can result in more efficient compression, in practice we used Huffman more frequently, as it offers improved decoding speed.

Temporal scheme
Our temporal scheme is applied on consecutive frames in a sequence to efficiently encode static objects in the scene, such as buildings and the ground. A point is marked as temporal if it exactly matches a point in the previous frame in terms of location and attributes. However, by allowing some flexibility in the attributes, especially the colour ones, more points can be encoded as temporal. In that case, we need to signal the differences (deltas) between the temporal points and their matching points at the previous frame.

For instance, as objects move throughout the sequence of frames, they may cast shadows over the otherwise static points, slightly altering their colour attributes. Other attributes, such as the normal, do not change between frames and so no differences need to be encoded. So that the decoder knows which points to copy from the previous frame, a bitarray is created with a length equal to the number of points in the previous frame, where each bit corresponds to a single point and signals whether the point should be copied to the current frame when decoding. The points copied from the previous frame only have differences in the colour attributes, which are encoded in a similar fashion to the attribute values, where they are quantized, sparsified, and entropy encoded.

Bitstream format

Our bitstream format contains all the necessary information to decode a point cloud frame, and it is comprised of a few main data layers. First, there is the fixed-size header layer, containing the number of points within the point cloud frame, information about the colour space and attributes encoded, the sizes of each of the following sections of the bitstream, and some information showing the pre-processing methods needed to decode the points.

Following this layer, there are four more sections of the bitstream: (1) the temporal section, (2) the attribute headers section, (3) the point position section, and (4) the attribute payload section.

Temporal Section
The temporal section contains a temporal bitarray that shows which points are encoded in temporal mode and which in intra mode. It also contains correction residuals that need to be applied to the temporal mode points during decoding, as well as the reference points from the previous frame.

In particular, the temporal section begins with a header, including the number of temporal points, the size of the payload, the attributes with non-zero temporal differences, and information about the run-length encoded bitarray mentioned above (length of the compressed array in bytes, and number of actual array values).

After the header, the temporal header contains a payload part: For each of the attributes that need a correction, the metadata and payload sizes are stored. The temporal header concludes with a metadata and payload part: the bitstream includes the entropy coding metadata for the temporal bitarray and its payload, followed by the metadata and payload for each of the attributes with non-zero differences.

Attribute headers section
This section contains information about the encoding method used for each attribute. Each attribute has a separate header which contains the pre-processing method that is used, the quantisation parameter (when necessary), the data average, the data transform type, and entropy coding metadata. Sparsification information follows, including the sparsification group size and the entropy encoding metadata for the sparsification bitarray.

Point position section
This part contains the data for decoding the position of each point. This data may be structured as either an octree or as separately encoded point coordinates (XYZ coding). After decoding, both methods provide the non-empty cubes with a given side length (which is always a power of two).

In the case of the octree, it first optionally contains the number of points within each cube at a given level of the octree that can be used for separating the attribute values to groups with different averages. Then it contains the number of octree codes that are included in the full octree, and the entropy encoding metadata and payload so that they can be decoded. Next in the bitstream is how many points are in each of the final level octree cubes.

The optional part of the octree section can be calculated while decoding the octree but is included to increase decoding speed, while it also allows the size of the groups to not exactly match the number of points in a level of the octree.

In the case of the separately encoded point coordinates (XYZ coding), first the bitarray contains the method used to encode the coordinates, such as differential coding, how the values were split into high and low bits, and how duplicate values of the high bits were handled. It then contains an array of sizes for each coordinate giving the metadata and payload sizes, finally followed by the metadata and payloads.

Finally, for both methods the bitarray contains for each point a fixed length code to encode where in each final level cube the points are.

Attributes payloads section
This part contains the entropy encoded payloads for each attribute.

EXPERIMENTAL RESULTS

Implementation details

We implemented the encoder and decoder in C++. To improve the encoding and decoding performance, we utilized multi-threading and IntelTM Single-Instruction-Multiple-Data (SIMD) intrinsics. We also designed and implemented a codec API, which was integrated in

the PresenZ VR workflow and was used in our subsequent tests on content provided by PresenZ.

Compression efficiency

The PresenZ VR system originally used the BLOSC [8] algorithm to compress the 6DoF point cloud data. BLOSC bitrates on the testing content averaged 457 MB/s (Megabytes per second) at 25 fps but could peak much higher for complex images.

When compressed using our encoder, the average bitrate was reduced to 93 MB/s, and peak rate to only 210 MB/s thus reducing the bitrate to well within the capabilities of conventional Solid State Drives (SSD).

The gain data compression compared to BLOSC depends on the similarity of the points between frames. For example, on one sequence with minor changes between frames the original BLOSC compressed size was 242MB/s and the new compressed size only 0.7MB/s. For a sequence with a lot of movement of points between each frame the original size was 130MB/s and the new compressed size 42MB/s. Our encoder was operated in a variable bitrate, constant quality mode. Detailed results for several VR sequences taken from the “Contstruct” VR movie [13] are shown on Table 1.

Computational efficiency

Our point cloud decoder needed to operate at 30 fps, to allow smooth viewing of the rendered VR movie. The following results were obtained on a laptop with an Intel i7-10875H CPU with 32 GB 3200MHz RAM. Our experiments show that we can achieve around 4-5 fps for encoding (using a non-optimized encoder) and 40+ fps when decoding. For rendering, we used a NVIDIA GeForce RTX 2080 Super 8 GB graphics card. For our experiments we used an Oculus Quest 2 VR headset.

The decoding speed is highly dependent on the encoded bitrate, which is dependent on the scene complexity, as illustrated in Table 1.

Sequence	Number of frames	BLOSC MB/s	Compressed MB/s	Max achievable decoding fps
902	1400	747	106	106
903	830	429	70	129
904	642	794	210	54
905	1038	409	109	104
906	580	163	20	160
907	853	523	126	96
908	1005	456	149	84
909	1105	630	70	129
910	1312	180	49	160
1000	480	159	47	156
Credits	258	5.4	4	135
Total	**9503**	**457**	**93**	**118**

CONCLUSIONS

The primary aim of developing a 6 DOF VR point cloud compression codec to enable the playback of 6 DOF VR movies generated by Presenz VR on high end gaming PCs was met, and the Construct VR volumetric movie was made publicly available [13].

Using the knowledge gained from developing the VC-6 (SMPTE ST 2117) and LCEVC (MPEG-5 Part 2) codecs and adopting an innovative approach to exploiting temporal redundancy between point-cloud frames we were able to compress the 6 DOF VR movie to average data rates under 100 MB/s whilst maintaining the quality required to provide a compelling VR experience, and doing so in such a way that the decoding and rendering can be performed by typical high end gaming laptop and desktop machines.

The VR compression codec we have developed provides a flexible framework for further innovation and is capable of being tuned to address other VR use cases. Future work includes enhancing and automating the encoding mode decisions to provide even greater compression performance, and further optimization of the encoder and decoder implementations.

In addition to the distribution of 6 DOF VR movies, other potential use cases include virtual production, medical imaging, and of course the metaverse.

REFERENCES

1. Barnard D. 2019. Degrees of Freedom (DoF): 3-DoF vs 6-DoF for VR Headset Selection. https://virtualspeech.com/blog/degrees-of-freedom-vr

2. Degrees of freedom (mechanics) April 2022
https://en.wikipedia.org/wiki/Degrees_of_freedom_(mechanics)

3. Solid State Drive. April 2022
https://en.wikipedia.org/wiki/Solid-state_drive

4. Streaming SIMD Extensions April 2022
https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

5. Intel Intrinsics Guide April 2022
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

6. Presenz VR March 2022
https://www.presenzvr.com/

7. Schnabel R.and Klein R., 2006 Octree-based Point-Cloud Compression
https://diglib.eg.org/xmlui/bitstream/handle/10.2312/SPBG.SPBG06.111-120/111-120.pdf?sequence=1

8. BLOSC In Depth April 2022
https://www.blosc.org/pages/blosc in-depth

9. D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki, and A. Tabatabai, “An overview of ongoing point cloud compression standardization activities: video-based (V-PCC) and geometry-based (G-PCC),” APSIPA Transactions on Signal and Information Processing, vol. 9, 2020.
https://mpeg-pcc.org/index.php/publications/an-overview-of-ongoing-point-cloud-compression-standardization-activities-video-based-v-pcc-and-geometry-based-g-pcc/

10. Thompson S. April 2020. Motion Sickness in VR: Why it happens and how to minimise it https://virtualspeech.com/blog/motion-sickness-vr?ref=footer

11. Normal (geometry)
https://en.wikipedia.org/wiki/Normal_(geometry)

12. Octree. March 2010 https://en.wikipedia.org/wiki/Octree#/media/File:Octree2.svg Licence: https://creativecommons.org/licenses/by-sa/3.0/

13. Construct VR – The volumetric movie – available on steam
https://store.steampowered.com/app/1674620/Construct_VR__The_Volumetric_Movie/

14. G-PCC codec description, MPEG 3D Graphics Coding, ISO/IEC JTC 1/SC 29/WG 7 N 00271, Serial Number 21244, January 2022.

15. Google Draco

16. SMPTE ST 2117-1 VC-6

17. MPEG-5 Part 2 LCEVC

18. Mekuria, R.N, Blom, C.L, & César Garcia, P.S. (2017). Design, implementation and evaluation of a point cloud codec for Tele-Immersive Video. IEEE Transactions on Circuits and Systems for Video Technology, 27(4), 828–842. doi:10.1109/TCSVT.2016.2543039

ACKNOWLEDGEMENTS

The authors would like to thank Tristan Salome and Jeroen De Coninck from Presenz for their help and support in the development and integration of the codec into the Presenz VR workflow and for the use of their 6 DOF VR movie content.