# DYNAMIC POLYGON CLOUD COMPRESSION

***(MICROSOFT RESEARCH TECHNICAL REPORT MSR-TR-2016-59)***

*Eduardo Pavez [1], Philip A. Chou[2], Ricardo L. de Queiroz[3], and Antonio Ortega[1]*

[1]University of Southern California, Los Angeles, CA, USA
[2]Microsoft Research, Redmond, WA, USA
[3]Universidade de Brasilia, Brasilia, Brazil

## ABSTRACT

We introduce the *polygon cloud*, also known as a polygon set or *soup*, as a compressible representation of 3D geometry (including its attributes, such as color texture) intermediate between polygonal meshes and point clouds. Dynamic or time-varying polygon clouds, like dynamic polygonal meshes and dynamic point clouds, can take advantage of temporal redundancy for compression, if certain challenges are addressed. In this paper, we propose methods for compressing both static and dynamic polygon clouds, specifically triangle clouds. We compare triangle clouds to both triangle meshes and point clouds in terms of compression, for live captured dynamic colored geometry. We find that triangle clouds can be compressed nearly as well as triangle meshes, while being far more robust to noise and other structures typically found in live captures, which violate the assumption of a smooth surface manifold, such as lines, points, and ragged boundaries. We also find that triangle clouds can be used to compress point clouds with significantly better performance than previously demonstrated point cloud compression methods.

***Index Terms—*** Polygon soup, dynamic mesh, point cloud, augmented reality, motion compensation, compression, graph transform, octree

## 1. INTRODUCTION

With the advent of virtual and augmented reality comes the birth of a new medium: live captured 3D content that can be experienced from any point of view. Such content ranges from static scans of compact 3D objects, to dynamic captures of non-rigid objects such as people, to captures of rooms including furniture, public spaces swarming with people, and whole cities in motion. For such content to be captured at one place and delivered to another for consumption by a virtual or augmented reality device (or by more conventional means), the content needs to be represented and compressed for transmission or storage. Applications include gaming, tele-immersive communication, free navigation of highly produced entertainment as well as live events, historical artifact and site preservation, acquisition for special effects, and so forth. This paper presents a novel means of representing and compressing the visual part of such content.

Until this point, two of the more promising approaches to representing both static and time-varying 3D scenes have been polygonal meshes and point clouds, along with their associated color information. However, both approaches have drawbacks. Polygonal meshes represent surfaces very well, but they are not robust to noise and other structures typically found in live captures, such as lines, points, and ragged boundaries that violate the assumptions of a smooth surface manifold. Point clouds, on the other hand, have a hard time modeling surfaces as compactly as meshes.

We propose a hybrid between polygonal meshes and point clouds: polygon clouds. Polygon clouds are sets of polygons, often called a polygon soup. The polygons in a polygon cloud are not required to represent a coherent surface. Like the points in a point cloud, the polygons in a polygon cloud can represent noisy, real-world geometry captures without any assumption of a smooth 2D manifold. In fact, any polygon in a polygon cloud can be collapsed into a point or line as a special case. The polygons may also overalap. On the other hand, the polygons in the cloud can also be stitched together into a watertight mesh if desired to represent a smooth surface.

For concreteness we focus on triangles instead of arbitrary polygons, and we develop an encoder and decoder for sequences of triangle clouds. We assume a simple group of frames (GOF) model, where each group of frames begins with an Intra (I) frame, also called a reference frame or a key frame, which is followed by a sequence of Predicted (P) frames, also called inter frames. The triangles are assumed to be consistent across frames. That is, the triangles' vertices are assumed to be tracked from one frame to the next. The trajec-

E. Pavez is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA, e-mail: pavezcar@usc.edu

P. A. Chou is with Microsoft Research, Redmond, WA, USA, e-mail: pachou@ieee.org.

R. L. de Queiroz is with the Computer Science Department at Universidade de Brasilia, Brasilia, Brazil, e-mail: queiroz@ieee.org.

Antonio Ortega is with the Department of Electrical Engineering at the University of Southern California, Los Angeles, CA, USA, e-mail: antonio.ortega@sipi.usc.edu

tories of the vertices are not constrained. Thus the triangles may change from frame to frame in location, orientation, and proportion. For geometry encoding, redundancy in the vertex trajectories is removed by a spatial othogonal transform followed by temporal prediction, allowing low latency. For color encoding, the triangles in each frame are projected back to the coordinate system of the reference frame. In the reference frame we voxelize the triangles in order to ensure that their color textures are sampled uniformly in space regardless of the sizes of the triangles, and in order to construct a common vector space in which to describe the color textures and their evolution from frame to frame. Redundancy of the color vectors is removed by a spatial orthogonal transform followed by temporal prediction, similar to redundancy removal for geometry. Uniform scalar quantization and entropy coding matched to the spatial transform are employed for both color and geometry.

We compare triangle clouds to both triangle meshes and point clouds in terms of compression, for live captured dynamic colored geometry. We find that triangle clouds can be compressed nearly as well as triangle meshes, while being far more flexible in representing live captured content. We also find that triangle clouds can be used to compress point clouds with significantly better performance than previously demonstrated point cloud compression methods.

The organization of the paper is as follows. Following a summary of related work in Section 2, preliminary material is presented in Section 3. The core of our compression system is presented in Section 5, while experimental results are presented in Section 6. The discussion and conclusion is in Section 9.

## 2. RELATED WORK

### 2.1. Mesh compression

3D mesh compression has a rich history, particularly from the 1990s forward. Overviews may be found in [1, 2, 3]. Fundamental is the need to code mesh topology, or connectivity, such as in [4, 5]. Beyond coding connectivity, coding the geometry, i.e., the positions of the vertices, is also fundamental. Many approaches have been taken, but one significant and practical approach to geometry coding is based on "geometry images" [6] and their temporal extension, "geometry videos" [7]. In these approaches, the mesh is partitioned into patches, the patches are projected onto a 2D plane as *charts*, non-overlapping charts are laid out in a rectangular *atlas*, and the atlas is compressed using a standard image or video coder, compressing both the geometry and the texture (i.e., color) data. For dynamic geometry, the meshes are assumed to be temporally consistent (i.e., connectivity is constant frame-to-frame) and the patches are likewise temporally consistent. Geometry videos have been used for representing and compressing free-viewpoint video of human actors [8]. Other key papers on mesh compression of human actors in the context of tele-immersion include [9, 10].

### 2.2. Motion estimation

A critical part of dynamic mesh compression is the ability to track points over time. If a mesh is defined for a keyframe, and the vertices are tracked over subsequent frames, then the mesh becomes a temporally consistent dynamic mesh. There is a huge body of literature in the 3D tracking, 3D motion estimation or scene flow, 3D interest point detection and matching, 3D correspondence, non-rigid registration, and the like. We are particularly influenced by [11, 12, 13], all of which produce in real time, given data from one or more RGBD sensors for every frame $t$, a parameterized mapping $f_{\theta_t} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that maps points in frame $t$ to points in frame $t+1$. Though corrections may need to be made at each frame, chaining the mappings together over time yields trajectories for any given set of points. Compressing these trajectories is similar to compressing motion capture (mocap) trajectories, which has been well studied. [14] is a recent example with many references. Compression typically involves an intra-frame transform to remove spatial redundancy and either temporal prediction (if low latency is required) or a temporal transform (if the entire clip or group of frames is available) to remove temporal redundancy, as in [15].

### 2.3. Graph signal processing

Graph Signal Processing (GSP) has emerged as an extension of the theory of linear shift invariant signal processing to the processing of signals on discrete graphs, where the shift operator is taken to be the adjacency matrix of the graph, or alternatively the Laplacian matrix of the graph [16, 17]. GSP was extended to critically sampled perfect reconstuction wavelet filter banks in [18, 19]. These constructions were used for dynamic mesh compression in [20, 21].

### 2.4. Point cloud compression using octrees

Sparse Voxel Octrees (SVOs) were developed in the 1980s to represent the geometry of three-dimensional objects [22, 23]. Recently SVOs have been shown to have highly efficient implementations suitable for encoding at video frame rates [24]. In the guise of occupancy grids, they have also had significant use in robotics [25, 26, 27]. Octrees were first used for point cloud compression in [28]. They were further developed for progressive point cloud coding, including color attribute compression, in [29]. Octrees were extended to coding of dynamic point clouds (i.e., point cloud sequences) in [30]. The focus of [30] was geometry coding; their color attribute coding remained rudimentary. Their method of inter-frame geometry coding was to take the exclusive-OR (XOR) between frames and code the XOR using an octree. Their method was implemented in the Point Cloud Library [31].

### 2.5. Color attribute compression for static point clouds

To better compress the color attributes in *static* voxelized point clouds, Zhang, Florencio, and Loop used transform coding based on the Graph Fourier Transform (GFT), recently developed in the theory of Graph Signal Processing [32]. While transform coding based on the GFT has good compression performance, it requires eigen-decompositions for each coded block, and hence may not be computationally attractive. To improve the computational efficiency, while not sacrificing compression performance, Queiroz and Chou developed an orthogonal Region-Adaptive Hierarchical Transform (RAHT) along with an entropy coder [33]. RAHT is essentially a Haar transform with the coefficients appropriately weighted to take the non-uniform shape of the domain (or region) into account. As its structure matches the Sparse Voxel Octree, it is extremely fast to compute. Other approaches to non-uniform regions include the shape-adaptive DCT [34] and color palette coding [35]. Further approaches based on non-uniform sampling of an underlying stationary process can be found in [36], which uses the KLT matched to the sample, and in [37], which uses sparse representation and orthogonal matching pursuit.

### 2.6. Dynamic point cloud compression

Thanou, Chou, and Frossard [38, 39] were the first to deal fully with *dynamic* voxelized points clouds, by finding matches between points in adjacent frames, warping the previous frame to the current frame, predicting the color attributes of the current frame from the quantized colors of the previous frame, and coding the residual using the GFT-based method of [40]. Thanou et al. used the XOR-based method of Kammerl et al. [30] for inter-frame geometry compression. However, the method of [30] proved to be inefficient, in a rate-distortion sense, for anything except slowly moving subjects, for two reasons. First, the method "predicts" the current frame from the previous frame, without any motion compensation. Second, the method codes the geometry losslessly, and so has no ability to perform a rate-distortion trade-off. To address these shortcomings, Queiroz and Chou [41] used block-based motion compensation and rate-distortion optimization to select between coding modes (intra or motion-compensated coding) for each block. Further, they applied RAHT to coding the color attributes (in intra-frame mode), color prediction residuals (in inter-frame mode), and the motion vectors (in inter-frame mode). They also used in-loop deblocking filters. Mekuria et al. [42] independently proposed block-based motion compensation for dynamic point cloud sequences. Although they did not use rate-distortion optimization, they used affine transformations for each motion-compensated block, rather than just translations. Unfortunately, it appears that block-based motion compensation of dynamic point cloud geometry tends to produce gaps between blocks, which are perceptually more damaging than indicated by some objective metrics, such as the Haussdorf-based metrics commonly used in geometry compression [43].

### 2.7. Key learnings

Some of the key learnings from the previous work, taken as a whole, are that

- Point clouds are preferable to meshes for resilience to noise and non-manifold signals measured in real world signals, especially for real time capture where the computational cost of heavy duty pre-processing (e.g., surface reconstruction, topological denoising, charting) can be prohibitive.
- For geometry coding in static scenes, point clouds appear to be more compressible than meshes, even though the performance of point cloud geometry coding seems to be limited by the lossless nature of the current octree methods. In addition, octree processing for geometry coding is extremely fast.
- For color attribute coding in static scenes, both point clouds and meshes appear to be well compressible. If charting is possible, compressing the color as an image may win out due to the maturity of image compression algorithms today. However, direct octree processing for color attribute coding is extremely fast, as it is for geometry coding.
- For both geometry and color attribute coding in dynamic scenes (or inter-frame coding), temporally consistent dynamic meshes are highly compressible. However, finding a temporally consistent mesh can be challenging from a topological point of view as well as from a computational point of view.

In our work, we aim to achieve the high compression efficiency possible with intra-frame point cloud compression and inter-frame dynamic mesh compression, while simultaneously achieving the high computational efficiency possible with octree-based processing, as well as its robustness to real-world noise and non-manifold data.

## 3. SYSTEM OVERVIEW

### 3.1. Notation

Denote the set of integers from 1 to $N$ by $[N]$. Sets will be denoted using calligraphic fonts and matrices and vectors using bold fonts.

### 3.2. Dynamic triangle clouds

A dynamic triangle cloud is a numerical representation of a time changing 3D scene or object. We denote it by a sequence

| symbol | description |
|---|---|
| $[N]$ | set of integers $\{1, 2, \cdots, N\}$ |
| $t$ | time or frame index |
| $v_i$ or $v_i^{(t)}$ | 3D point with coordinates $x_i, y_i, z_i$ |
| $f_m$ or $f_m^{(t)}$ | face with vertex indices $i_m, j_m, k_m$ |
| $c_n$ or $c_n^{(t)}$ | color with components $Y_n, U_n, V_n$ |
| $a_i$ or $a_i^{(t)}$ | generic attribute vector $a_{i1}, \ldots, a_{in}$ |
| $\mathcal{V}$ or $\mathcal{V}^{(t)}$ | set of $N_p$ points $\{v_1, \ldots, v_{N_p}\}$ |
| $\mathcal{F}$ or $\mathcal{F}^{(t)}$ | set of $N_f$ faces $\{f_1, \ldots, f_{N_f}\}$ |
| $\mathcal{C}$ or $\mathcal{C}^{(t)}$ | set of $N_c$ colors $\{c_1, \ldots, c_{N_c}\}$ |
| $\mathcal{A}$ or $\mathcal{A}^{(t)}$ | set of $N_a$ attributes $\{a_1, \ldots, a_{N_a}\}$ |
| $\mathcal{T}$ or $\mathcal{T}^{(t)}$ | triangle cloud $(\mathcal{V}, \mathcal{F}, \mathcal{C})$ |
| $\mathcal{P}$ or $\mathcal{P}^{(t)}$ | point cloud $(\mathcal{V}, \mathcal{A})$ |
| $\mathbf{V}$ or $\mathbf{V}^{(t)}$ | $N_p \times 3$ matrix with $i$-th row $[x_i, y_i, z_i]$ |
| $\mathbf{F}$ or $\mathbf{F}^{(t)}$ | $N_f \times 3$ matrix with $m$-th row $[i_m, j_m, k_m]$ |
| $\mathbf{C}$ or $\mathbf{C}^{(t)}$ | $N_c \times 3$ matrix with $n$-th row $[Y_n, U_n, V_n]$ |
| $\mathbf{A}$ | list (i.e., matrix) of attributes |
| $\mathbf{TA}$ | list of transformed attributes |
| $\mathbf{M}$,$\mathbf{M}_v$,$\mathbf{M}_1$ | lists of Morton codes |
| $\mathbf{W}$,$\mathbf{W}_v$,$\mathbf{W}_{rv}$ | lists of weights |
| $\mathbf{I}, \mathbf{I}_v, \mathbf{I}_{rv}$ | lists of indices |
| $\hat{\mathbf{V}}$,$\hat{\mathbf{C}}$,$\hat{\mathbf{A}}$,... | lists of quantized or reproduced quantities |
| $\hat{\mathbf{V}}_v$ or $\hat{\mathbf{V}}_v^{(t)}$ | list of voxelized vertices |
| $\mathbf{V}_r$ | list of refined vertices |
| $\hat{\mathbf{V}}_{rv}$ or $\hat{\mathbf{V}}_{rv}^{(t)}$ | list of voxelized refined vertices |
| $\mathbf{C}_r = \mathbf{C}$ | list of colors of refined vertices |
| $\mathbf{C}_{rv}$ or $\mathbf{C}_{rv}^{(t)}$ | list of colors of voxelized refined vertices |
| $J$ | octree depth |
| $U$ | upsampling factor |
| $\Delta_{motion}$ | motion quantization stepsize |
| $\Delta_{color,intra}$ | intra-frame color quantization stepsize |
| $\Delta_{color,inter}$ | inter-frame color quantization stepsize |

**Table 1**: Notation

$\{\mathcal{T}^{(t)}\}$ where $\mathcal{T}^{(t)}$ is a triangle cloud at time $t$. Each individual frame $\mathcal{T}^{(t)}$ has geometry (shape and position) and color information.

The geometry information consists of a list of vertices $\mathcal{V}^{(t)} = \{v_i^{(t)} : i = 1, \cdots, N_p\}$, where each vertex $v_i^{(t)} = [x_i^{(t)}, y_i^{(t)}, z_i^{(t)}]$ is a point in 3D, and a list of triangles (or faces) $\mathcal{F}^{(t)} = \{f_m^{(t)} : m = 1, \cdots, N_f\}$, where each face $f_m^{(t)} = [i_m^{(t)}, j_m^{(t)}, k_m^{(t)}]$ is a vector of indices of vertices from $\mathcal{V}^{(t)}$. We denote by $\mathbf{V}^{(t)}$ the $N_p \times 3$ matrix whose $i$th row is the point $v_i^{(t)}$, and similarly we denote by $\mathbf{F}^{(t)}$ the $N_f \times 3$ matrix whose $m$th row is the triangle $f_m^{(t)}$. The triangles in a triangle cloud do not have to be adjacent or form a mesh, and they can overlap. Two or more vertices of a triangle may have the same coordinates, thus collapsing into a line or point.

The color information consists of a list of colors $\mathcal{C}^{(t)} = \{c_n^{(t)} : n = 1, \cdots, N_c\}$, where each color $c_n^{(t)} = [Y_n^{(t)}, U_n^{(t)}, V_n^{(t)}]$ is a vector in YUV space (or other convenient color space). We denote by $\mathbf{C}^{(t)}$ the $N_c \times 3$ matrix whose $n$th row is the color $c_n^{(t)}$. The list of colors represents the colors across the surfaces of the triangles. To be specific, $c_n^{(t)}$ is the color of a "refined" vertex $v_r^{(t)}(n)$, where the refined vertices are obtained by uniformly subdividing each triangle in $\mathcal{F}^{(t)}$ by upsampling factor $U$, as shown in Figure 1b for $U = 4$. We denote by $\mathbf{V}_r^{(t)}$ the $N_c \times 3$ matrix whose $n$th row is the refined vertex $v_r^{(t)}(n)$. $\mathbf{V}_r^{(t)}$ can be computed from $\mathcal{V}^{(t)}$ and $\mathcal{F}^{(t)}$, so we do not need to encode it, but we will use it to compress the color information. Note that $N_c = N_f(U+1)(U+2)/2$. The upsampling factor $U$ should be high enough so that it does not limit the color spatial resolution obtainable by the color cameras. In our experiments, we set $U = 10$ or higher. Setting $U$ higher does not typically affect the bit rate significantly, though it does affect memory and computation in the encoder and decoder.

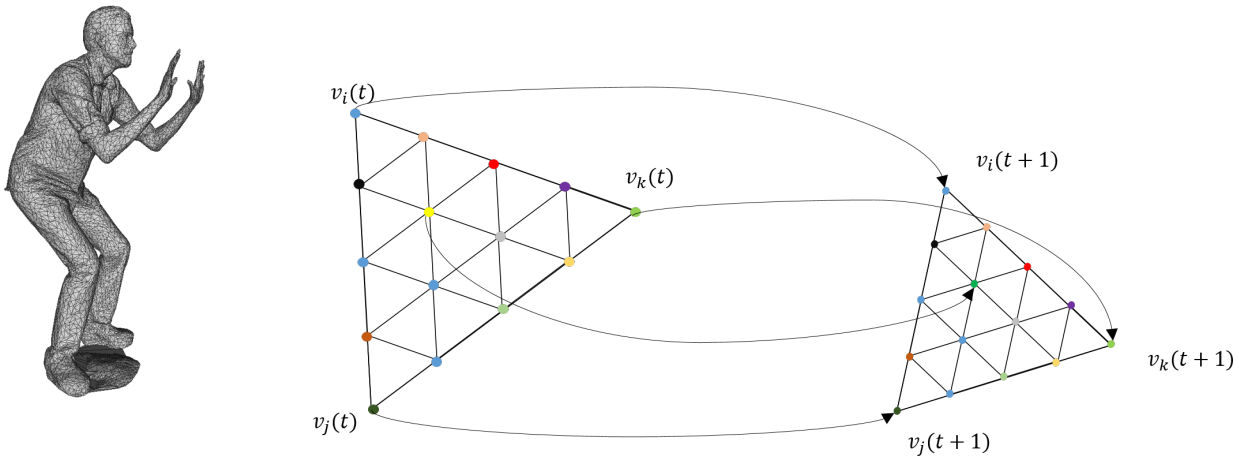
Thus frame $t$ can be represented by the triple $\mathbf{V}^{(t)}$, $\mathbf{F}^{(t)}$, $\mathbf{C}^{(t)}$. We use a Group of Frames (GOF) model, in which the sequence is partitioned into GOFs. The GOFs are processed independently. Without loss of generality, we label the frames in a GOF $t = 1 \ldots, N$. There are two types of frames: reference and predicted. In each GOF, the first frame ($t = 1$) is a reference frame and all other frames ($t = 2, \ldots, N$) are predicted. Within a GOF, all frames must have the same number of vertices, triangles, and colors: $\forall t \in [N]$, $\mathbf{V}^{(t)} \in \mathbb{R}^{N_p \times 3}$, $\mathbf{F}^{(t)} \in [N_p]^{N_f \times 3}$ and $\mathbf{C}^{(t)} \in \mathbb{R}^{N_c \times 3}$. The triangles are assumed to be consistent across frames so that there is a correspondence between colors and vertices within the GOF. In Figure 1b we show an example of the correspondences between two consecutive frames in a GOF. Across GOFs, the GOFs may have a different numbers of frames, vertices, triangles, and colors.

In the following two subsections, we outline how to obtain a triangle cloud from an existing point cloud or an existing triangular mesh.

#### *3.2.1. Converting a dynamic point cloud to a dynamic triangle cloud*

A dynamic point cloud is a sequence of point clouds $\{\mathcal{P}^{(t)}\}$, where each $\mathcal{P}^{(t)}$ is a list of $[x, y, z]$ coordinates with an attribute attached to it like color. To produce a triangle cloud, we need a way to fit a point cloud to a set of triangles in such a way that we produce GOFs with consistent triangles. One way of doing that is the following.

1. Decide if frame in $\mathcal{P}^{(t)}$ is reference or predicted.
2. If reference frame:
   (a) Fit triangles to point cloud to obtain $\mathbf{V}, \mathbf{F}$, where $\mathbf{V}$ is a list of vertices and $\mathbf{F}$ is a list of triangles.




(a) "Man" mesh.

(b) Correspondences between two consecutive frames.

**Fig. 1**

(b) Subdivide each triangle, and project each vertex of the subdivision to the closest point in the cloud to obtain $\mathbf{C}$.

3. If predicted frame:

   (a) Deform triangle cloud of previous reference frame to fit point cloud to obtain $\mathbf{V}$.

   (b) Subdivide each triangle, and project each vertex of the subdivision to the closest point in the cloud to obtain $\mathbf{C}$.

   (c) Go to step 1.

This process will introduce geometric distortion and a change in the number of points. All points will be forced to lie in a uniform grid on the surface of a triangle. The triangle fitting can be done using triangular mesh fitting and tracking techniques such as in [11, 12, 13].

#### *3.2.2. Converting a dynamic triangular mesh to a dynamic triangle cloud*

The geometry of a triangular mesh is represented by a list of key points or vertices and their connectivity, given by an array of 3D coordinates $\mathbf{V}$ and faces $\mathbf{F}$. The triangles are constrained to form a smooth surface without holes. For color, the mesh representation typically includes an array of 2D texture coordinates $\mathbf{T} \in \mathbb{R}^{N_p \times 2}$ and a texture image. The color at any point on a face can be retrieved (for rendering) by interpolating the texture coordinates at that point on the face and sampling the image at the interpolated coordinates. The sequence of triangular meshes is assumed to be temporally consistent, meaning that within a GOF, the meshes of the predicted frames are deformations of the reference frame. The sizes and positions of the triangles may change but the deformed mesh still represents a smooth surface. The sequence of key points $\mathbf{V}^{(t)}$ thus can be traced from frame to frame and the faces are all the same. To convert the color information into the dynamic triangle cloud format, for each frame and each triangle, the mesh sub-division function can be applied to obtain texture coordinates of refined triangles. Then the texture image can be sampled and a color matrix $\mathbf{C}$ can be formed for each frame.

### 3.3. Compression system overview

In this section we provide an overview of our system for compressing dynamic triangle clouds. We compress consecutive GOFs sequentially and independently, so we focus on the system for compressing an individual GOF $(\mathbf{V}^{(t)}, \mathbf{F}^{(t)}, \mathbf{C}^{(t)})$ for $t \in [N]$.

For the reference frame, we voxelize the vertices $\mathbf{V}^{(1)}$, and then encode the voxelized vertices $\mathbf{V}_v^{(1)}$ using octree encoding. We encode the connectivity $\mathbf{F}^{(1)}$ with a lossless entropy coder. (We could use method such as EdgeBreaker or TFAN [4, 5], but for simplicity for this small amount of data we use the lossless universal encoder *gzip*.) We code the connectivity only once per GOF (i.e., for the reference frame), since the connectivity is consistent across the GOF, i.e., $\mathbf{F}^{(t)} = \mathbf{F}^{(1)}$ for $t \in [N]$. We voxelize the colors $\mathbf{C}^{(1)}$, and encode the voxelized colors $\mathbf{C}_{rv}^{(1)}$ using a transform coding method that combines the region adaptive hierarchical transform (RAHT) [33], uniform scalar quantization, and adaptive Run-Length Golomb-Rice (RLGR) entropy coding [44]. At the cost of additional complexity, the RAHT transform could be replaced by transforms with higher performance [36, 37].

For predicted frames, we compute prediction residuals

from the previously decoded frame. Specifically, for each predicted frame $t > 1$ we compute a motion residual $\Delta\mathbf{V}_v^{(t)} = \mathbf{V}_v^{(t)} - \hat{\mathbf{V}}_v^{(t-1)}$ and a color residual $\Delta\mathbf{C}_{rv}^{(t)} = \mathbf{C}_{rv}^{(t)} - \hat{\mathbf{C}}_{rv}^{(t-1)}$, where we have denoted with a *hat* a quantity that has been compressed and decompressed. These residuals are encoded using again RAHT followed by uniform scalar quantization and entropy coding.

It is important to note that we do not directly compress the list of vertices $\mathbf{V}^{(t)}$ or the the list of colors $\mathbf{C}^{(t)}$ (or their prediction residuals). Rather, we voxelize them first *with respect to their corresponding vertices in the reference frame*, and then compress them. This ensures that 1) if two or more vertices or colors fall into the same voxel, they receive the same representation and hence are encoded only once, and 2) the colors (on the set of refined vertices) are resampled uniformly in space regardless of the density of triangles.

In the next section, we describe the basic elements of the system: refinement, voxelization, octrees, and transform coding. In the section after that, we describe in detail how these basic elements are put together to encode and decode a sequence of triangle clouds.

## 4. REFINEMENT, VOXELIZATION, OCTREES, AND TRANSFORM CODING

### 4.1. Refinement

Given a list of faces $\mathbf{F}$, its corresponding list of vertices $\mathbf{V}$, and upsampling factor $U$, a list of "refined" vertices $\mathbf{V}_r$ can be produced using Algorithm 1. Step 1 (in Matlab notation) assembles three equal-length lists of vertices (each as an $N_f \times 3$ matrix), containing the three vertices of every face. Step 5 appends a linear combinations of the faces' vertices to a growing list of refined vertices.

**Algorithm 1** Refinement (*refine*)

**Input:** $\mathbf{V}$, $\mathbf{F}$, $U$

1: $\mathbf{V}_i = \mathbf{V}(\mathbf{F}(:,i),:)$, $i = 1,2,3$ // $i$th vertex of all faces
2: Initialize $k = U$ and $\mathbf{V}_r =$ empty list
3: **for** $i = 0$ to $U$ **do**
4: **for** $j = 0$ to $k$ **do**
5: $\mathbf{V}_r = [\mathbf{V}_r; \mathbf{V}_1 + (\mathbf{V}_2 - \mathbf{V}_1)i/U + (\mathbf{V}_3 - \mathbf{V}_1)j/U]$
6: **end for**
7: $k = k - 1$
8: **end for**

**Output:** $\mathbf{V}_r$

We assume that the list of colors $\mathbf{C}$ is in 1-1 correspondence with the list of refined vertices $\mathbf{V}_r$. Indeed, to obtain the colors $\mathbf{C}$ from a textured mesh, the 2D texture coordinates $\mathbf{T}$ can be linearly interpolated in the same manner as the 3D position coordinates $\mathbf{V}$ to obtain "refined" texture coordinates $\mathbf{T}_r$ which may then be used to lookup appropriate color $\mathbf{C}_r = \mathbf{C}$ in the texture map.

### 4.2. Morton codes and voxelization

A *voxel* is a volumetric element used to represent the attributes of of an object in 3D over a small region of space. Analogous to 2D pixels, 3D voxels are defined on a uniform grid. We assume the geometric data live in the unit cube $[0,1)^3$, and we uniformly partition the cube into voxels of size $2^{-J} \times 2^{-J} \times 2^{-J}$.

Now consider a list of points $\mathbf{V} = [v_i]$ and an equal-length list of attributes $\mathbf{A} = [a_i]$, where $a_i$ is the real-valued attribute (or vector of attributes) of $v_i$. (These may be, for example, the list of refined vertices $\mathbf{V}_r$ and their associated colors $\mathbf{C}_r = \mathbf{C}$ as discussed above.) In the process of *voxelization*, the points are partitioned into voxels, and the attributes associated with the points in a voxel are averaged. The points within each voxel are quantized to the voxel center. Each occupied voxel is then represented by the voxel center and the average of the attributes of the points in the voxel. Moreover, the occupied voxels are put into Z-scan order, also known as Morton order [45]. The first step in voxelization is to quantize the vertices and to produce their Morton codes. The Morton code $m$ for a point $(x, y, z)$ is obtained simply by interleaving (or "swizzling") the bits of $x$, $y$, and $z$, with $x$ being lower order than $y$, and $y$ being lower order than $z$. For example, if $x = x_4x_2x_1$, $y = y_4y_2y_1$, and $z = z_4z_2z_1$ (written in binary), then the Morton code for the point would be $m = z_4y_4x_4z_4y_4x_4z_1y_1x_1$. The Morton codes are sorted, duplicates are removed, and all attributes whose vertices have a particular Morton code are averaged.

The procedure is detailed in Algorithm 2. $\mathbf{V}_{int}$ is the list of vertices with their coordinates, previously in $[0,1)$, now mapped to integers in $\{0,\ldots,2^J-1\}$. $\mathbf{M}$ is the corresponding list of Morton codes. $\mathbf{M}_v$ is the list of Morton codes, sorted with duplicates removed, using the Matlab function *unique*. $\mathbf{I}$ and $\mathbf{I}_v$ are vectors of indices such that $\mathbf{M}_v = \mathbf{M}(\mathbf{I})$ and $\mathbf{M} = \mathbf{M}_v(\mathbf{I}_v)$, in Matlab notation. (That is, the $i_v$th element of $\mathbf{M}_v$ is the $\mathbf{I}(i_v)$th element of $\mathbf{M}$ and the $i$th element of $\mathbf{M}$ is the $\mathbf{I}_v(i)$th element of $\mathbf{M}_v$.) $\mathbf{A}_v = [\bar{a}_j]$ is the list of attribute averages

$$\bar{a}_j = \frac{1}{N_j} \sum_{i:\mathbf{M}(i)=\mathbf{M}_v(j)} a_i, \tag{1}$$

where $N_j$ is the number of elements in the sum. $\mathbf{V}_v$ is the list of voxel centers. The algorithm has complexity $\mathcal{O}(N \log N)$, where $N$ is the number of input vertices.

### 4.3. Octree encoding

Any set of voxels in the unit cube, each of size $2^{-J} \times 2^{-J} \times 2^{-J}$, designated *occupied* voxels, can be represented with an octree of depth $J$ [22, 23]. An octree is a recursive subdivision of a cube into smaller cubes, as illustrated in Figure 2. Cubes are subdivided only as long as they are occupied (i.e., contain any occupied voxels). This recursive subdivision can

**Algorithm 2** Voxelization (*voxelize*)

**Input:** $\mathbf{V}$, $\mathbf{A}$, $J$

1: $\mathbf{V}_{int} = floor(\mathbf{V} * 2^J)$ // map coords to $\{0, \ldots, 2^J - 1\}$
2: $\mathbf{M} = morton(\mathbf{V}_{int})$ // generate list of morton codes
3: $[\mathbf{M}_v, \mathbf{I}, \mathbf{I}_v] = unique(\mathbf{M})$ // find unique codes, and sort
4: $\mathbf{A}_v = [\bar{a}_j]$, where $\bar{a}_j = mean(\mathbf{A}(\mathbf{M} = \mathbf{M}_v(j))$ is the average of all attributes whose Morton code is the $j$th Morton code in the list $\mathbf{M}_v$
5: $\mathbf{V}_v = (\mathbf{V}_{int}(\mathbf{I}, :) + 0.5) * 2^{-j}$ // compute voxel centers

**Output:** $\mathbf{V}_v$ (or equivalently $\mathbf{M}_v$), $\mathbf{A}_v$, $\mathbf{I}_v$.

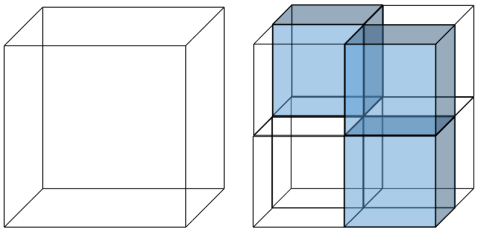


**Fig. 2**: Cube subdivision. Blue cubes represent occupied regions of space.

be represented by an octree with depth $J$, where the root corresponds to the unit cube. The leaves of the tree correspond to the set of occupied voxels.

There is a close connection between octrees and Morton codes. In fact, the Morton code of a voxel, which has length $3J$ bits broken into $J$ binary triples, encodes the path in the octree from the root to the leaf containing the voxel. Moreover, the sorted list of Morton codes results from a depth-first traversal of the tree.

Each internal node of the tree can be represented by one byte, to indicate which of its eight children are occupied. If these bytes are serialized in a pre-order traversal of the tree, the serialization (which has a length in bytes equal to the number of internal nodes of the tree) can be used as a description of the octree, from which the octree can be reconstructed. Hence the description can also be used to encode the ordered list of Morton codes of the leaves. This description can be further compressed using a context adaptive arithmetic encoder. However, for simplicity in our experiments, we use *gzip* instead of an arithmetic encoder.

In this way, we encode any set of occupied voxels in a canonical (Morton) order.

### 4.4. Transform coding

In this section we describe the region adaptive hierarchical transform (RAHT) [33] and its efficient implementation. RAHT can be described as a sequence of orthonormal transforms applied to attribute data living on the leaves of an octree. For simplicity we assume the attributes are scalars. This transform processes voxelized attributes in a bottom up fashion, starting at the leaves of the octree. The inverse transform reverses this order.

Consider eight adjacent voxels, three of which are occupied, having the same parent in the octree, as shown in Figure 3. The colored voxels are occupied (have an attribute) and the transparent ones are empty. Each occupied voxel is assigned a unit weight. Fot the forward transform, transformed attribute values and weights will be propagated up the tree.

One level of the forward transform proceeds as follows. Pick a direction $(x, y, z)$, then check whether there are two occupied cubes that can be processed along that direction. In the leftmost part of Figure 3 there are only three occupied cubes, *red*, *yellow*, and *blue*, having weights $w_r$, $w_y$, and $w_b$, respectively. To process in the direction of the $x$ axis, since the *blue* cube does not have a neighbor along the horizontal direction, we copy its attribute value $a_b$ to the second stage and keep its weight $w_b$. The attribute values $a_y$ and $a_r$ of the *yellow* and *red* cubes can be processed together using the orthonormal transformation

$$\begin{bmatrix} a_g^0 \\ a_g^1 \end{bmatrix} = \frac{1}{\sqrt{w_y + w_r}} \begin{bmatrix} \sqrt{w_y} & \sqrt{w_r} \\ -\sqrt{w_r} & \sqrt{w_y} \end{bmatrix} \begin{bmatrix} a_y \\ a_r \end{bmatrix}, \qquad (2)$$

where the transformed coefficients $a_g^0$ and $a_g^1$ respectively represent low pass and high pass coefficients appropriately weighted. Both transform coefficients now represent information from a region with weight $w_g = w_y + w_r$ (*green* cube). The high pass coefficient is stored for entropy coding along with its weight, while the low pass coefficient is further processed and put in the *green* cube. For processing along the $y$ axis, the *green* and *blue* cubes do not have neighbors, so their values are copied to the next level. Then we process in the $z$ direction using the same transformation in (2) with weights $w_g$ and $w_b$.

This process is repeated for each cube of eight subcubes at each level of the octree. After $J$ levels, there remains one low pass coefficient that corresponds to the DC component; the remainder are high pass coefficients. Since after each processing of a pair of coefficients, the weights are added and used during the next transformation, the weights can be interpreted as being inversely proportional to frequency. The DC coefficient is the one that has the largest weight, as it is processed more times and represents information from the entire cube, while the high pass coefficients, which are produced earlier, have smaller weights because they contain information from a smalle region. The weights depend only on the octree (not the coefficients themselves), and thus can provide a frequency ordering for the coefficients. We sort the transformed coefficients by decreasing magnitude of weight.

Finally, the sorted coefficients are quantized using uniform scalar quantization, and are entropy coded using adaptive Run Length Golomb-Rice coding [44].

Efficient implementations of RAHT and its inverse are detailed in Algorithms 4 and 5, respectively. Algorithm 3 is a

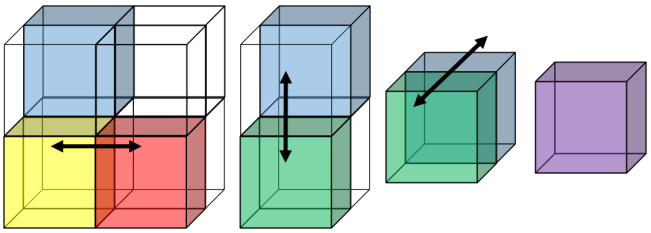


**Fig. 3**: One level of RAHT applied to a cube of eight voxels, three of which are occupied.

prologue to each. Algorithm 6 is our uniform scalar quantization.

**Algorithm 3** prologue to Region Adaptive Hierarchical Transform (RAHT) and its Inverse (IRAHT) (*prologue*)

**Input:** $\mathbf{V}$, $J$

$\mathbf{M}_1 = morton(\mathbf{V})$ // morton codes
$N = length(\mathbf{M})$ // number of points
**for** $\ell = 1$ to $3J$ **do** // define $(\mathbf{I}_\ell, \mathbf{M}_\ell, \mathbf{W}_\ell, \mathbf{F}_\ell), \forall \ell$
  **if** $\ell = 1$ **then** // initialize indices of coeffs at layer 1
    $\mathbf{I}_1 = (1:N)^T$ // vector of indices from 1 to $N$
  **else** // define indices of coeffs at layer $\ell$
    $\mathbf{I}_\ell = \mathbf{I}_{\ell-1}(\neg[0; \mathbf{F}_{\ell-1}])$ // left sibs and singletons
  **end if**
  $\mathbf{M}_\ell = \mathbf{M}_1(\mathbf{I}_\ell)$ // morton codes at layer $\ell$
  $\mathbf{W}_\ell = [\mathbf{I}_\ell(2:end); N+1] - \mathbf{I}_\ell$ // weights
  $\mathbf{D} = \mathbf{M}_\ell(1:end-1) \oplus \mathbf{M}_\ell(2:end)$ // path diffs
  $\mathbf{F}_\ell = (\mathbf{D} \wedge (2^{3J} - 2^\ell)) \neq 0$ // left sibling flags
**end for**

**Output:** $\{(\mathbf{M}_\ell, \mathbf{I}_\ell, \mathbf{W}_\ell, \mathbf{F}_\ell) : \ell = 1, \ldots, 3J\}$, and $N$

## 5. ENCODING AND DECODING

In this section we describe in detail encoding and decoding of dynamic triangle clouds. First we describe encoding and decoding of reference frames. Following that, we describe encoding and decoding of predicted frames. For both reference and predicted frames, we describe first how geometry is encoded and decoded, and then how color is encoded and decoded. The overall system is shown in Figure 4.

### 5.1. Encoding and decoding of reference frames

For reference frames, encoding is summarized in Algorithm 7, while decoding is summarized in Algorithm 8.

#### *5.1.1. Geometry encoding and decoding*

We assume that the vertices in $\mathbf{V}^{(1)}$ are in Morton order. If not, we put them into Morton order and permute the indices in

**Algorithm 4** Region Adaptive Hierarchical Transform (*RAHT*)

**Input:** $\mathbf{V}$, $\mathbf{A}$, $J$

$[\{(\mathbf{M}_\ell, \mathbf{I}_\ell, \mathbf{W}_\ell, \mathbf{F}_\ell)\}, N] = prologue(\mathbf{V}, J)$
$\mathbf{TA} = \mathbf{A}$ // perform transform in place
$\mathbf{W} = \mathbf{1}$ // initialize to $N$-vector of unit weights
**for** $\ell = 1$ to $3J - 1$ **do**
  $\mathbf{i}_0 = \mathbf{I}_\ell([\mathbf{F}_\ell; 0] == 1)$ // left sibling indices
  $\mathbf{i}_1 = \mathbf{I}_\ell([0; \mathbf{F}_\ell] == 1)$ // right sibling indices
  $\mathbf{w}_0 = \mathbf{W}_\ell([\mathbf{F}_\ell; 0] == 1)$ // left sibling weights
  $\mathbf{w}_1 = \mathbf{W}_\ell([0; \mathbf{F}_\ell] == 1)$ // right sibling weights
  $\mathbf{x}_0 = \mathbf{TA}(\mathbf{i}_0, :)$ // left sibling coefficients
  $\mathbf{x}_1 = \mathbf{TA}(\mathbf{i}_1, :)$ // right sibling coefficients
  $\mathbf{a} = repmat(sqrt(\mathbf{w}_0./(\mathbf{w}_0+\mathbf{w}_1)), 1, size(\mathbf{TA}, 2))$
  $\mathbf{b} = repmat(sqrt(\mathbf{w}_1./(\mathbf{w}_0+\mathbf{w}_1)), 1, size(\mathbf{TA}, 2))$
  $\mathbf{TA}(\mathbf{i}_0, :) = \mathbf{a} .* \mathbf{x}_0 + \mathbf{b} .* \mathbf{x}_1$
  $\mathbf{TA}(\mathbf{i}_1, :) = -\mathbf{b} .* \mathbf{x}_0 + \mathbf{a} .* \mathbf{x}_1$
  $\mathbf{W}(\mathbf{i}_0) = \mathbf{W}(\mathbf{i}_0) + \mathbf{W}(\mathbf{i}_1)$
  $\mathbf{W}(\mathbf{i}_1) = \mathbf{W}(\mathbf{i}_0)$
**end for**

**Output:** $\mathbf{TA}$, $\mathbf{W}$

**Algorithm 5** Inverse Region Adaptive Hierarchical Transform (*IRAHT*)

**Input:** $\mathbf{V}$, $\mathbf{TA}$, $J$

$[\{(\mathbf{M}_\ell, \mathbf{I}_\ell, \mathbf{W}_\ell, \mathbf{F}_\ell)\}, N] = prologue(\mathbf{V}, J)$
$\mathbf{A} = \mathbf{TA}$ // perform inverse transform in place
**for** $\ell = 3J - 1$ down to 1 **do**
  $\mathbf{i}_0 = \mathbf{I}_\ell([\mathbf{F}_\ell; 0] == 1)$ // left sibling indices
  $\mathbf{i}_1 = \mathbf{I}_\ell([0; \mathbf{F}_\ell] == 1)$ // right sibling indices
  $\mathbf{w}_0 = \mathbf{W}_\ell([\mathbf{F}_\ell; 0] == 1)$ // left sibling weights
  $\mathbf{w}_1 = \mathbf{W}_\ell([0; \mathbf{F}_\ell] == 1)$ // right sibling weights
  $\mathbf{x}_0 = \mathbf{TA}(\mathbf{i}_0, :)$ // left sibling coefficients
  $\mathbf{x}_1 = \mathbf{TA}(\mathbf{i}_1, :)$ // right sibling coefficients
  $\mathbf{a} = repmat(sqrt(\mathbf{w}_0./(\mathbf{w}_0+\mathbf{w}_1)), 1, size(\mathbf{TA}, 2))$
  $\mathbf{b} = repmat(sqrt(\mathbf{w}_1./(\mathbf{w}_0+\mathbf{w}_1)), 1, size(\mathbf{TA}, 2))$
  $\mathbf{TA}(\mathbf{i}_0, :) = \mathbf{a} .* \mathbf{x}_0 - \mathbf{b} .* \mathbf{x}_1$
  $\mathbf{TA}(\mathbf{i}_1, :) = \mathbf{b} .* \mathbf{x}_0 + \mathbf{a} .* \mathbf{x}_1$
**end for**

**Output:** $\mathbf{A}$

**Algorithm 6** Uniform scalar quantization (*quantize*)

**Input:** $\mathbf{A}$, $step$

$\hat{\mathbf{A}} = round(\mathbf{A}/step) * step$

**Output:** $\hat{\mathbf{A}}$

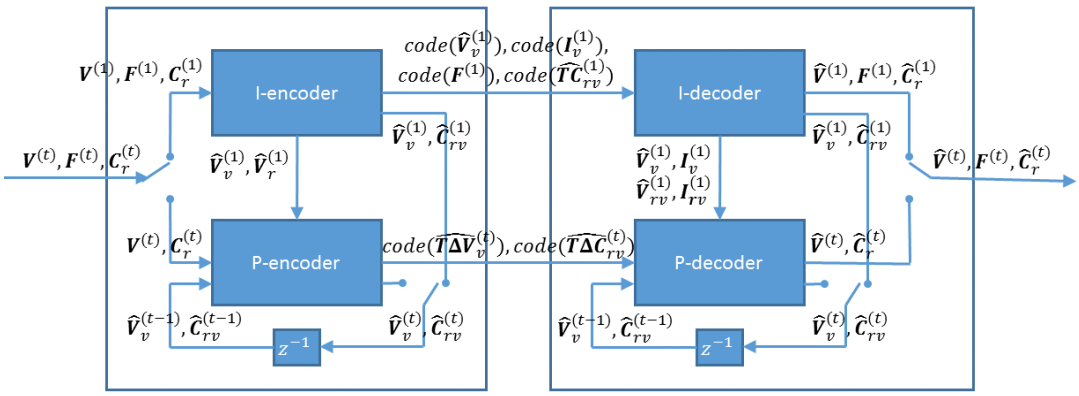



**Fig. 4**: Encoder (left) and decoder (right).

**Algorithm 7** Encode reference frame (I-encoder)

**Input:** $J, U, \Delta_{color,intra}$ (from system parameters)
**Input:** $\mathbf{V}^{(1)}$, $\mathbf{F}^{(1)}$, $\mathbf{C}_r^{(1)}$ (from system input)
// Geometry
$\hat{\mathbf{V}}^{(1)} = quantize(\mathbf{V}^{(1)}, 2^{-J})$
$[\hat{\mathbf{V}}_v^{(1)}, \mathbf{I}_v^{(1)}] = voxelize(\hat{\mathbf{V}}^{(1)}, J)$ s.t. $\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$
// Color
$\hat{\mathbf{V}}_r^{(1)} = refine(\hat{\mathbf{V}}^{(1)}, \mathbf{F}^{(1)}, U)$
$[\hat{\mathbf{V}}_{rv}^{(1)}, \mathbf{C}_{rv}^{(1)}, \mathbf{I}_{rv}^{(1)}] = voxelize(\hat{\mathbf{V}}_r^{(1)}, \mathbf{C}_r^{(1)}, J)$ s.t. $\hat{\mathbf{V}}_r^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$
$[\mathbf{TC}_{rv}^{(1)}, \mathbf{W}_{rv}^{(1)}] = RAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \mathbf{C}_{rv}^{(1)}, J)$
$\widehat{\mathbf{TC}}_{rv}^{(1)} = quantize(\mathbf{TC}_{rv}^{(1)}, \Delta_{color,intra})$
$\hat{\mathbf{C}}_{rv}^{(1)} = IRAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \widehat{\mathbf{TC}}_{rv}^{(1)}, J)$
**Output:** $code(\mathbf{V}_v^{(1)})$, $code(\mathbf{I}_v^{(1)})$, $code(\mathbf{F}^{(1)})$, $code(\widehat{\mathbf{TC}}_{rv}^{(1)})$ (to reference frame decoder)
**Output:** $\hat{\mathbf{V}}_v^{(1)}$, $\hat{\mathbf{V}}_r^{(1)}$ (to predicted frame encoder)
**Output:** $\hat{\mathbf{V}}_v^{(1)}$, $\hat{\mathbf{C}}_{rv}^{(1)}$ (to reference frame buffer)

**Algorithm 8** Decode reference frame (I-decoder)

**Input:** $J, U, \Delta_{color,intra}$ (from system parameters)
**Input:** $code(\mathbf{V}_v^{(1)})$, $code(\mathbf{I}_v^{(1)})$, $code(\mathbf{F}^{(1)})$, $code(\widehat{\mathbf{TC}}_{rv}^{(1)})$ (from reference frame encoder)
// Geometry
$\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$
// Color
$\hat{\mathbf{V}}_r^{(1)} = refine(\hat{\mathbf{V}}^{(1)}, \mathbf{F}^{(1)}, U)$
$[\hat{\mathbf{V}}_{rv}^{(1)}, \mathbf{I}_{rv}^{(1)}] = voxelize(\hat{\mathbf{V}}_r^{(1)}, J)$ s.t. $\hat{\mathbf{V}}_r^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$
$\mathbf{W}_{rv}^{(1)} = RAHT(\hat{\mathbf{V}}_{rv}^{(1)}, J)$
$\hat{\mathbf{C}}_{rv}^{(1)} = IRAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \widehat{\mathbf{TC}}_{rv}^{(1)}, J)$
$\hat{\mathbf{C}}_r^{(1)} = \hat{\mathbf{C}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$
**Output:** $\hat{\mathbf{V}}^{(1)}$, $\mathbf{F}^{(1)}$, $\hat{\mathbf{C}}_r^{(1)}$ (to renderer)
**Output:** $\hat{\mathbf{V}}_v^{(1)}$, $\mathbf{I}_v^{(1)}$, $\hat{\mathbf{V}}_{rv}^{(1)}$, $\mathbf{I}_{rv}^{(1)}$ (to predicted frame decoder)
**Output:** $\hat{\mathbf{V}}_v^{(1)}$, $\hat{\mathbf{C}}_{rv}^{(1)}$ (to reference frame buffer)

$\mathbf{F}^{(1)}$ accordingly. The lists $\mathbf{V}^{(1)}$ and $\mathbf{F}^{(1)}$ are the geometry-related quantities in the reference frame transmitted from the encoder to the decoder. $\mathbf{V}^{(1)}$ will be reconstructed at the decoder with some loss as $\hat{\mathbf{V}}^{(1)}$, and $\mathbf{F}^{(1)}$ will be reconstructed losslessly. We now describe the process.

At the encoder, the vertices in $\mathbf{V}^{(1)}$ are first quantized to the voxel grid, producing a list of quantized vertices $\hat{\mathbf{V}}^{(1)}$, the same length as $\mathbf{V}^{(1)}$. There may be duplicates in $\hat{\mathbf{V}}^{(1)}$, because some vertices may have collapsed to the same grid point. $\hat{\mathbf{V}}^{(1)}$ is then voxelized (without attributes), the effect of which is simply to remove the duplicates, producing a possibly slightly shorter list $\hat{\mathbf{V}}_v^{(1)}$ along with a list of indices $\mathbf{I}_v^{(1)}$ such that (in Matlab notation) $\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$. Since $\hat{\mathbf{V}}_v^{(1)}$ has no duplicates, it represents a *set* of voxels. This set can be described by an octree. The byte sequence representing the octree can be compressed with any entropy encoder; we use *gzip*. The list of indices $\mathbf{I}_v^{(1)}$, which has the same length as $\hat{\mathbf{V}}^{(1)}$, indicates, essentially, how to restore the duplicates, which are missing from $\hat{\mathbf{V}}_v^{(1)}$. In fact, the indices in $\mathbf{I}_v^{(1)}$ increase in unit steps for all vertices in $\hat{\mathbf{V}}^{(1)}$ except the duplicates, for which there is no increase. The list of indices is thus a sequence of runs of unit increases alternating with runs of zero increases. This binary sequence of increases can be encoded with any entropy encoder; we use *gzip* on the run lengths. Finally the list of faces $\mathbf{F}^{(1)}$ can be encoded with any entropy encoder; we again use *gzip*, though algorithms such as [4, 5] might also be used.

The decoder entropy decodes $\hat{\mathbf{V}}_v^{(1)}$, $\mathbf{I}_v^{(1)}$, and $\mathbf{F}^{(1)}$, and then recovers $\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$, which is the quantized version of $\mathbf{V}^{(1)}$, to obtain both $\hat{\mathbf{V}}^{(1)}$ and $\mathbf{F}^{(1)}$.

### 5.1.2. *Color encoding and decoding*

Let $\mathbf{V}_r^{(1)} = refine(\mathbf{V}^{(1)}, \mathbf{F}^{(1)}, U)$ be the list of "refined vertices" obtained by upsampling, by factor $U$, the faces $\mathbf{F}^{(1)}$ whose vertices are $\mathbf{V}^{(1)}$. We assume that the colors in the list $\mathbf{C}_r^{(1)} = \mathbf{C}^{(1)}$ correspond to the refined vertices in $\mathbf{V}_r^{(1)}$. In particular, the lists have the same length. Here, we subscript

the list of colors by an 'r' to indicate that it corresponds to the list of refined vertices.

When the vertices $\mathbf{V}^{(1)}$ are quantized to $\hat{\mathbf{V}}^{(1)}$, the refined vertices change to $\hat{\mathbf{V}}_r^{(1)} = refine(\hat{\mathbf{V}}^{(1)}, \mathbf{F}^{(1)}, U)$. The list of colors $\mathbf{C}_r^{(1)}$ can also be considered as indicating the colors on $\hat{\mathbf{V}}_r^{(1)}$. The list $\mathbf{C}_r^{(1)}$ is the color-related quantity in the reference frame transmitted from the encoder to the decoder. The decoder will reconstruct $\mathbf{C}_r^{(1)}$ with some loss $\hat{\mathbf{C}}_r^{(1)}$. We now describe the process.

At the encoder, the refined vertices $\hat{\mathbf{V}}_r^{(1)}$ are obtained as described above. Then the vertices $\hat{\mathbf{V}}_r^{(1)}$ and their associated color attributes $\mathbf{C}_r^{(1)}$ are voxelized to obtain a list of voxels $\hat{\mathbf{V}}_{rv}^{(1)}$, the list of voxel colors $\mathbf{C}_{rv}^{(1)}$, and the list of indices $\mathbf{I}_{rv}^{(1)}$ such that (in Matlab notation) $\hat{\mathbf{V}}_r^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$. The list of indices $\mathbf{I}_{rv}^{(1)}$ has the same length as $\hat{\mathbf{V}}_r^{(1)}$, and contains for each vertex in $\hat{\mathbf{V}}_r^{(1)}$ the index of its corresponding vertex in $\hat{\mathbf{V}}_{rv}^{(1)}$. As there may be many refined vertices falling into each voxel, the list $\hat{\mathbf{V}}_{rv}^{(1)}$ may be significantly shorter than the list $\hat{\mathbf{V}}_r^{(1)}$ (and the list $\mathbf{I}_{rv}^{(1)}$). However, unlike the geometry case, in this case the list $\mathbf{I}_{rv}^{(1)}$ need not be transmitted.

The list of voxel colors $\hat{\mathbf{C}}_{rv}^{(1)}$, each with unit weight, is transformed by RAHT to an equal-length list of transformed colors $\mathbf{TC}_{rv}^{(1)}$ and associated weights $\mathbf{W}_{rv}^{(1)}$. The transformed colors then quantized with stepsize $intraColorStep$ to obtain $\widehat{\mathbf{TC}}_{rv}^{(1)}$. The quantized RAHT coefficients are entropy coded by the method suggested in [33], and transmitted. Finally, $\widehat{\mathbf{TC}}_{rv}^{(1)}$ is inverse transformed by RAHT using the associated weights to obtain $\hat{\mathbf{C}}_{rv}^{(1)}$. These represent the quantized voxel colors, and will be used as a reference for subsequent predicted frames.

At the decoder, similarly, the refined vertices $\hat{\mathbf{V}}_r^{(1)}$ are obtained by upsampling, by factor $U$, the faces $\mathbf{F}^{(1)}$ whose vertices are $\hat{\mathbf{V}}^{(1)}$ (both of which have been decoded already in the geometry step). $\hat{\mathbf{V}}_r^{(1)}$ is then voxelized (without attributes) to produce the list of voxels $\hat{\mathbf{V}}_{rv}^{(1)}$ and list of indices $\mathbf{I}_{rv}^{(1)}$ such that $\hat{\mathbf{V}}_r^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$. The weights $\mathbf{W}_{rv}^{(1)}$ are recovered by using RAHT to transform a random signal on the vertices $\hat{\mathbf{V}}_r^{(1)}$, each with unit weight. Then $\widehat{\mathbf{TC}}_{rv}^{(1)}$ is entropy decoded and inverse transformed by RAHT using the recovered weights to obtain the quantized voxel colors $\hat{\mathbf{C}}_{rv}^{(1)}$. Finally, the quantized refined vertex colors can be obtained as $\hat{\mathbf{C}}_r^{(1)} = \hat{\mathbf{C}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$.

### 5.2. Encoding and decoding of predicted frames

We assume that all $N$ frames in a GOP are aligned. That is, the lists of faces, $\mathbf{F}^{(1)}, \ldots, \mathbf{F}^{(N)}$, are all identical. Moreover, the lists of vertices, $\mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}$, all correspond in the sense that the $i$th vertex in list $\mathbf{V}^{(1)}$ (say, $\mathbf{v}^{(1)}(i)$) corresponds to the $i$th vertex in list $\mathbf{V}^{(t)}$ (say, $\mathbf{v}^{(t)}(i)$), for all $t = 1, \ldots, N$. $(\mathbf{v}^{(1)}(i), \ldots, \mathbf{v}^{(N)}(i))$ is the trajectory of vertex $i$ over the GOF, $i = 1, \ldots, N_p$, where $N_p$ is the number of vertices.

Similarly, when the faces are upsampled by factor $U$ to create new lists of refined vertices, $\mathbf{V}_r^{(1)}, \ldots, \mathbf{V}_r^{(N)}$ — and their colors, $\mathbf{C}_r^{(1)}, \ldots, \mathbf{C}_r^{(N)}$ — the $i_r$th elements of these lists also correspond to each other across the GOF, $i_r = 1, \ldots, N_c$, where $N_c$ is the number of refined vertices, or the number of colors.

The trajectory $\{(\mathbf{v}^{(1)}(i), \ldots, \mathbf{v}^{(N)}(i)) : i = 1, \ldots, N_p\}$ can be considered an attribute of vertex $\mathbf{v}^{(1)}(i)$, and likewise the trajectories $\{(\mathbf{v}_r^{(1)}(i_r), \ldots, \mathbf{v}_r^{(N)}(i_r)) : i_r = 1, \ldots, N_c\}$ and $\{(\mathbf{C}_r^{(1)}(i_r), \ldots, \mathbf{C}_r^{(N)}(i_r)) : i_r = 1, \ldots, N_c\}$ can be considered attributes of refined vertex $\mathbf{v}_r^{(1)}(i_r)$. Thus the trajectories can be partitioned according to how the vertex $\mathbf{v}^{(1)}(i)$ and the refined vertex $\mathbf{v}_r^{(1)}(i_r)$ are voxelized. As for any attribute, the average of the trajectories in each cell of the partition is used to represent all trajectories in the cell. Our scheme codes these representative trajectories. This could be a problem if trajectories diverge from the same, or nearly the same, point, for example, when clapping hands separate. However, this situation is usually avoided by retarting the GOF by inserting a key frame, or reference frame, whenever the topology changes, and by using a sufficiently fine voxel grid.

In this section we show how to encode and decode the predicted frames, i.e., frames $t = 2, \ldots, N$, in each GOF. The frames are processed one at a time, with no look-ahead, to minimize latency. The encoding is detailed in Algorithm 9, while decoding is detailed in Algorithm 10.

#### *5.2.1. Geometry encoding and decoding*

At the encoder, for frame $t$, as for frame 1, the vertices $\mathbf{V}^{(1)}$, or equivalently the vertices $\hat{\mathbf{V}}^{(1)}$, are voxelized. However, for frame $t > 1$ the voxelization occurs with attributes $\mathbf{V}^{(t)}$. As for frame 1, this produces a possibly slightly shorter list $\hat{\mathbf{V}}_v^{(1)}$ along with a list of indices $\mathbf{I}_v^{(1)}$ such that $\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$. In addition, it produces an equal-length list of representative attributes, $\mathbf{V}_v^{(t)}$. Such a list is produced every frame. Therefore the previous frame can be used as a prediction. The prediction residual $\Delta\mathbf{V}_v^{(t)} = \mathbf{V}_v^{(t)} - \hat{\mathbf{V}}_v^{(t-1)}$ is transformed, quantized (with stepsize $\Delta_{motion}$), inverse transformed, and added to the prediction to obtain the reproduction $\hat{\mathbf{V}}_v^{(t)}$, which goes into the frame buffer. The quantized transform coefficients are entropy coded. We use adaptive RLGR as the entropy coder.

At the decoder, the entropy code for the quantized transform coefficients of the prediction residual is received, entropy decoded, inverse transformed, inverse quantized, and added to the prediction to obtain $\hat{\mathbf{V}}_v^{(t)}$, which goes into the frame buffer. Finally $\hat{\mathbf{V}}^{(t)} = \hat{\mathbf{V}}_v^{(t)}(\mathbf{I}_v^{(1)})$ is sent to the renderer.

**Algorithm 9** Encode predicted frame (P-encoder)

**Input:** $J$, $\Delta_{motion}$, $\Delta_{color,inter}$ (from system parameters)
**Input:** $\mathbf{V}^{(t)}$, $\mathbf{C}_r^{(t)}$ (from system input)
**Input:** $\hat{\mathbf{V}}^{(1)}$, $\hat{\mathbf{V}}_r^{(1)}$ (from reference frame encoder)
**Input:** $\hat{\mathbf{V}}_v^{(t-1)}$, $\hat{\mathbf{C}}_{rv}^{(t-1)}$ (from previous frame buffer)

// Geometry
$[\hat{\mathbf{V}}_v^{(1)}, \mathbf{V}_v^{(t)}, \mathbf{I}_v^{(1)}] = voxelize(\hat{\mathbf{V}}^{(1)}, \mathbf{V}^{(t)}, J)$ s.t. $\hat{\mathbf{V}}^{(1)} = \hat{\mathbf{V}}_v^{(1)}(\mathbf{I}_v^{(1)})$
$\Delta\mathbf{V}_v^{(t)} = \mathbf{V}_v^{(t)} - \hat{\mathbf{V}}_v^{(t-1)}$
$[\mathbf{T}\Delta\mathbf{V}_v^{(t)}, \mathbf{W}_v^{(1)}] = RAHT(\hat{\mathbf{V}}_v^{(1)}, \Delta\mathbf{V}_v^{(t)}, J)$
$\widehat{\mathbf{T}\Delta\mathbf{V}}_v^{(t)} = quantize(\mathbf{T}\Delta\mathbf{V}_v^{(t)}, \Delta_{motion})$
$\widehat{\Delta\mathbf{V}}_v^{(t)} = IRAHT(\hat{\mathbf{V}}_v^{(1)}, \widehat{\mathbf{T}\Delta\mathbf{V}}_v^{(t)}, J)$
$\hat{\mathbf{V}}_v^{(t)} = \hat{\mathbf{V}}_v^{(t-1)} + \widehat{\Delta\mathbf{V}}_v^{(t)}$
// Color
$[\hat{\mathbf{V}}_{rv}^{(1)}, \mathbf{C}_{rv}^{(t)}, \mathbf{I}_{rv}^{(1)}] = voxelize(\hat{\mathbf{V}}_r^{(1)}, \mathbf{C}_r^{(t)}, J)$ s.t. $\hat{\mathbf{V}}_r^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$
$\Delta\mathbf{C}_{rv}^{(t)} = \mathbf{C}_{rv}^{(t)} - \hat{\mathbf{C}}_{rv}^{(t-1)}$
$[\mathbf{T}\Delta\mathbf{C}_{rv}^{(t)}, \mathbf{W}_{rv}^{(1)}] = RAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \Delta\mathbf{C}_{rv}^{(t)}, J)$
$\widehat{\mathbf{T}\Delta\mathbf{C}}_{rv}^{(t)} = quantize(\mathbf{T}\Delta\mathbf{C}_{rv}^{(t)}, \Delta_{color,inter})$
$\widehat{\Delta\mathbf{C}}_{rv}^{(t)} = IRAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \widehat{\mathbf{T}\Delta\mathbf{C}}_{rv}^{(t)}, J)$
$\hat{\mathbf{C}}_{rv}^{(t)} = \hat{\mathbf{C}}_{rv}^{(t-1)} + \widehat{\Delta\mathbf{C}}_{rv}^{(t)}$

**Output:** $code(\widehat{\mathbf{T}\Delta\mathbf{V}}_v^{(t)})$, $code(\widehat{\mathbf{T}\Delta\mathbf{C}}_{rv}^{(t)})$ (to predicted frame decoder)
**Output:** $\hat{\mathbf{V}}_v^{(t)}$, $\hat{\mathbf{C}}_{rv}^{(t)}$ (to previous frame buffer)

**Algorithm 10** Decode predicted frame (P-decoder)

**Input:** $J$, $U$, $\Delta_{motion}$, $\Delta_{color,inter}$ (from system parameters)
**Input:** $code(\widehat{\Delta\mathbf{V}}_v^{(t)})$, $code(\widehat{\mathbf{T}\Delta\mathbf{C}}_{rv}^{(t)})$ (from predicted frame encoder)
**Input:** $\hat{\mathbf{V}}_v^{(1)}$, $\mathbf{I}_v^{(1)}$, $\hat{\mathbf{V}}_{rv}^{(1)}$, $\mathbf{I}_{rv}^{(1)}$ (from reference frame decoder)
**Input:** $\hat{\mathbf{V}}_v^{(t-1)}$, $\hat{\mathbf{C}}_{rv}^{(t-1)}$ (from previous frame buffer)

// Geometry
$\mathbf{W}_v^{(1)} = RAHT(\hat{\mathbf{V}}_v^{(1)}, J)$
$\widehat{\Delta\mathbf{V}}_v^{(t)} = IRAHT(\hat{\mathbf{V}}_v^{(1)}, \widehat{\mathbf{T}\Delta\mathbf{V}}_v^{(t)}, J)$
$\hat{\mathbf{V}}_v^{(t)} = \hat{\mathbf{V}}_v^{(t-1)} + \widehat{\Delta\mathbf{V}}_v^{(t)}$
$\hat{\mathbf{V}}^{(t)} = \hat{\mathbf{V}}_v^{(t)}(\mathbf{I}_v^{(1)})$
// Color
$\mathbf{W}_{rv}^{(1)} = RAHT(\hat{\mathbf{V}}_{rv}^{(1)}, J)$
$\widehat{\Delta\mathbf{C}}_{rv}^{(t)} = IRAHT(\hat{\mathbf{V}}_{rv}^{(1)}, \widehat{\mathbf{T}\Delta\mathbf{C}}_{rv}^{(t)}, J)$
$\hat{\mathbf{C}}_{rv}^{(t)} = \hat{\mathbf{C}}_{rv}^{(t-1)} + \widehat{\Delta\mathbf{C}}_{rv}^{(t)}$
$\hat{\mathbf{C}}_r^{(t)} = \hat{\mathbf{C}}_{rv}^{(t)}(\mathbf{I}_{rv}^{(1)})$

**Output:** $\hat{\mathbf{V}}^{(t)}$, $\mathbf{F}^{(1)}$, $\hat{\mathbf{C}}_r^{(t)}$ (to renderer)
**Output:** $\hat{\mathbf{V}}_v^{(t)}$, $\hat{\mathbf{C}}_{rv}^{(t)}$ (to previous frame buffer)

| sequence | Nframes | fr/sec | $\lvert\mathbf{V}\rvert$ | $\lvert\mathbf{F}\rvert$ | $J$ | upsample |
|---|---|---|---|---|---|---|
| soccer<br>lipstick<br>yellow dress | | | | | | |

**Table 2**: HCap sequences, average values across all frames.

#### *5.2.2. Color encoding and decoding*

At the encoder, for frame $t > 1$, as for frame $t = 1$, the refined vertices $\hat{\mathbf{V}}_r^{(t)}$, are voxelized with attributes $\mathbf{C}_r^{(t)}$. As for frame $t = 1$, this produces a significantly shorter list $\hat{\mathbf{V}}_{rv}^{(1)}$ along with a list of indices $\mathbf{I}_{rv}^{(1)}$ such that $\hat{\mathbf{V}}_r{}^{(1)} = \hat{\mathbf{V}}_{rv}^{(1)}(\mathbf{I}_{rv}^{(1)})$. In addition, it produces a list of representative attributes, $\mathbf{C}_{rv}^{(t)}$. Such a list is produced every frame. Therefore the previous frame can be used as a prediction. The prediction residual $\Delta\mathbf{C}_{rv}^{(t)} = \mathbf{C}_{rv}^{(t)} - \hat{\mathbf{C}}_{rv}^{(t-1)}$ is transformed, quantized (with step-size $\Delta_{color,inter}$), inverse transformed, and added to the prediction to obtain the reproduction $\hat{\mathbf{C}}_{rv}^{(t)}$, which goes into the frame buffer. The quantized transform coefficients are entropy coded. We use adaptive RLGR as the entropy coder.

At the decoder, the entropy code for the quantized transform coefficients of the prediction residual is received, entropy decoded, inverse transformed, inverse quantized, and added to the prediction to obtain $\hat{\mathbf{C}}_{rv}^{(t)}$, which goes into the frame buffer. Finally $\hat{\mathbf{C}}_r^{(t)} = \hat{\mathbf{C}}_{rv}^{(t)}(\mathbf{I}_{rv}^{(1)})$ is sent to the renderer.

### 5.3. Back to triangle clouds

Since the output of the compression system recovers a sequence of voxelized geometry and voxel projected attributes, for visualization and distortion computation we need to reconstruct a triangle cloud. First we approximately invert the voxel projection by computing the minimum mean squared error reconstruction, wich can be done using the partitions of the voxel projection with respect to the vertices of the subdivided triangles. That procedure can be done for geometry and color in reference and predicted frames.

For visualization, we use the triangle subdivision function and construct a point cloud, whose points lie in the surfaces of triangles and have color attributes. That point cloud can be further refined using the same triangle division function and color interpolation to obtain a denser point cloud.

## 6. EXPERIMENTS

### 6.1. Dataset

Describe HCap dataset in general, what it has, and how we process it to get what we want. Describe pre-processing, including depth and upsampling parameters. Then put details of sequences in a table

### 6.2. Error metrics

Comparing 3D geometry poses some challenges because there is not an agreed upon metric or distortion measure for this type of data. We consider several metrics for both color and geometry to evaluate different aspects of our compression system.

#### *6.2.1. Transform coding distortion $D_0$*

Since within the encoder we are working with attributes projected onto voxels, and we have correspondences withing each GOFs because everything is projected onto the voxels of the reference frame. We have worked so far with several parameters: octree depth $J$ and upsampling factor $U$ are considered global parameters that depend more on the data acquisition and rendering and are fixed for our compression system. The functions that will introduce distortion are the `voxProj` function and quantization funciton $Q$ for color and motion in predicted frames. Since the voxel projection only depends on the voxel and triangle sizes, we cannot control its error within the encoder. Therefore for rate-distortion analysis we will analyze the errors between voxel projected color attributes before and after transform coding for reference frames, and for predicted frames we will analyze geometry and color errors before and after predictive transform coding. For geometry we use signal to noise ration (SNR) between an original and a compressed frame with geometry coordinate matrices $\mathbf{V}_{vox}$ and $\hat{\mathbf{V}}_{vox}$

$$SNR = -20\log_{10}\left(\frac{\|\hat{\mathbf{V}}_{vox} - \mathbf{V}_{vox}\|_F}{\|\mathbf{V}_{vox}\|_F}\right). \qquad (3)$$

For color we compute peak signal to noise ration PSNR for $YUV$ components separately. For color $Y$ attributes $\mathbf{C}^Y_{vox}$ $\hat{\mathbf{C}}^Y_{vox}$ before and after transform coding we compute

$$PSNR = -20\log_{10}\left(\frac{\|\hat{\mathbf{C}}^Y_{vox} - \mathbf{C}^Y_{vox}\|_2}{255\sqrt{N_c}}\right) \qquad (4)$$

#### *6.2.2. High resolution triangle cloud distortion $D_1$*

We re-sample the dynamic mesh dataset with a higher resolution $40$ instead of $10$ used for compression. We take our reconstructed dynamic triangle cloud, then compute the corresponding point cloud using the method from section 5.3, then subdivide the triangles by a factor of $4$ with color interpolation. The new pointcloud is much more dense and comparable with the higher resolution resampled dynamic mesh. We compute geometry SNR and color PSNR.

#### *6.2.3. Principal projections distortion $D_2$*

Orthogonal projection onto 6 faces of a cube. We obtain six images per frame. Since the projection will depend on the geometry compression, we will report only color PSNR but as a function of motion rate and color rate. This experiment will allows us to evalueate the effect of geometry compression on the color quality.

**Fig. 5**: Geometry SNR vs bits/voxel, soccer lipstick sequence

#### *6.2.4. 1-nearest-neighbor matching distortion $D_3$*

We will voxel project each frame with respect to its own geometry (opposed as how we have been doing it, with respect to the reference frame). We obtain a sequence of voxelized point clouds, we compare using the 1-nearest-neighbor matching to compute SNR and PSNR. Since the matching depends on the geometry, we will analyze the effect of geometry compression on color PSNR.

### 6.3. Geometry compression

For geometry compression the only variable within the encoder is $\Delta_{motion}$, we will show rates for parameter taking values $\{1, 2, 4, 8, 16, 32, 64\}$. And show the following information:

1. average geometry SNR vs bit/vox, whole sequence and for reference frames only and predicted frames only. This is before and after transform coding on voxelized delta vectors.
2. Now on high resolution rendered data $U = 40$ for input and $U = 10$ for voxel size and $U = 4$ for linear interpolation.
   (a) High resolution direct comparison, geometry SNR and color PSNR vs motion rate
   (b) Cube projection distance, color PSNR vs motion rate.
   (c) voxelized 1-nearest-neighbor matching distance, geometry SNR and color PSNR vs motion rate.
   (d) show in a table motion Step, mbits/sec, bit/vox for predicted frames, and bit/vox for reference frames. Also number of reference frames and predicted frames.

### 6.4. Color Compression

Two parameters, intra and inter colorSteps.

### 6.5. Error metrics

1. Voxelize everything with respect to itself, not the key frame, so I will have to rewrite the pre process function.
2. project into cube, then get 6 images and do image psnr.

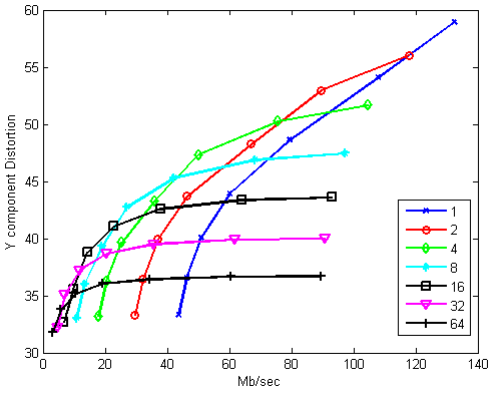



**Fig. 6**: Color Rate distortion curve for $Y$ component of *soccer* sequence

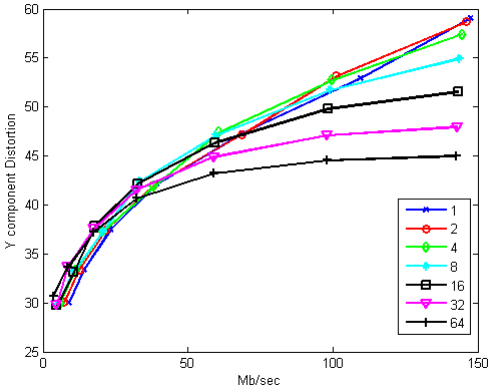



**Fig. 7**: Color Rate distortion curve for $Y$ component of *man* sequence

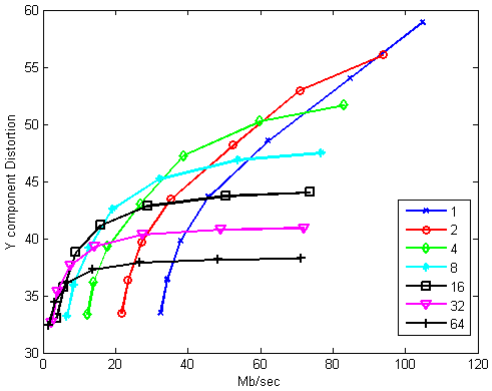



**Fig. 8**: Color Rate distortion curve for $Y$ component of *break dancer* sequence

3. for i-frames number of voxels will be the same, for origianl and reconstructed data, because we made it like that, for p frames, not necessarily. Compute nearest neighbor, and do distance, if the match is not 1 to 1, doesnt matter, just compute the number of edges in knn graph and divide by that when averaging.

## 7. EXPERIMENTS D0

First set of experiments, RD curves

1. plot color psnr vs geometry rates, for all pairs intra inter step, 1:64. One sequence average over all frames.

2. find a rule, e.g. intra  inter steps, then we will have only 1 color step parameter. This is done for all sequences, and we find this rule for all of them.

3. different sequences in the same color psnr vs color rate (one color rate) plot.

SEcond set of experiments things vs time

In the same plot put different sequences. Time plots

1. geometry MSE vs geometry rate plot so we can show that reference frames have zero error.

2. color psnr vs time

3. geom mse vs time

4. color rate vs time

5. geometry rate vs time

## 8. ICASSP EXPERIMENTS

- D0 experiments, RD curve for color, then fix one color parameter.

- RD for motion.

comparisons

- All intra: octtree on refined geometry+ raht on color. (150Mbits/sec)

- octree on coarse geometry +raht on color, this is our system in intra mode all the time.

- intra+inter, what we have

## 9. DISCUSSION AND CONCLUSION

## 10. EXTENSIONS AND IMPROVEMENTS

1. Better reference frame coding, intra prediction for color?
2. Generalized RAHT, less local, reduce artifacts.
3. Improve inter prediction for color and motion. Better key point tracker, maybe edge tracker for color, motion compensation. LOW COMPLEXITY COMPUTER VISION ALGOS.
4. Other entropy coders.
5. Other attributes like normals.
6. Add filtering to reduce artifacts.
7. RD optimization

## 11. REFERENCES


[1] P. Alliez and C. Gotsman, "Recent advances in compression of 3d meshes," in *Advances in Multiresolution for Geometric Modeling*, N. A. Dodgson, M. S. Floater, and M. A. Sabin, Eds., pp. 3–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[2] J. Peng, Chang-Su Kim, and C. C. Jay Kuo, "Technologies for 3d mesh compression: A survey," *Journal of Vis. Comun. and Image Represent.*, vol. 16, no. 6, pp. 688–733, Dec. 2005.

[3] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot, "3d mesh compression: survey, comparisons and emerging trends," *ACM Computing Surveys*, vol. 9, no. 4, 2013.

[4] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 5, no. 1, pp. 47–61, Jan. 1999.

[5] K. Mamou, T. Zaharia, and F. Prêteux, "TFAN: A low complexity 3d mesh compression algorithm," *Computer Animation and Virtual Worlds*, vol. 20, 2009.

[6] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe, "Geometry images," *ACM Trans. Graphics (SIGGRAPH)*, vol. 21, no. 3, pp. 355–361, July 2002.

[7] H. Briceño, P. Sander, L. McMillan, S. Gortler, and H. Hoppe, "Geometry videos: a new representation for 3d animations," in *Symp. Computer Animation*, 2003.

[8] A. Collet, M. Chuang, P. Sweeney, D. Gillett, D. Evseev, D. Calabrese, H. Hoppe, A. Kirk, and S. Sullivan, "High-quality streamable free-viewpoint video," *ACM Trans. Graphics (SIGGRAPH)*, vol. 34, no. 4, pp. 69:1–69:13, July 2015.

[9] R. Mekuria, M. Sanna, E. Izquierdo, D. C. A. Bulterman, and P. Cesar, "Enabling geometry-based 3-d tele-immersion with fast mesh compression and linear rateless coding," *IEEE Transactions on Multimedia*, vol. 16, no. 7, pp. 1809–1820, Nov 2014.

[10] A. Doumanoglou, D. S. Alexiadis, D. Zarpalas, and P. Daras, "Toward real-time and efficient compression of human time-varying meshes," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 12, pp. 2099–2116, Dec 2014.

[11] R. A. Newcombe, D. Fox, and S. M. Seitz, "Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 343–352.

[12] M. Dou, J. Taylor, H. Fuchs, A. Fitzgibbon, and S. Izadi, "3d scanning deformable objects with a single rgbd sensor," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 493–501.

[13] M. Dou, S. Khamis, Y. Degtyarev, P. Davidson, S. R. Fanello, A. Kowdle, S. Orts Escolano, C. Rhemann, D. Kim, J. Taylor, P. Kohli, V. Tankovich, and S. Izadi, "Fusion4d: real-time performance capture of challenging scenes," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 114, 2016.

[14] J. Hou, L. P. Chau, N. Magnenat-Thalmann, and Y. He, "Human motion capture data tailored transform coding," *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, no. 7, pp. 848–859, July 2015.

[15] J. Hou, L.-P. Chau, N. Magnenat-Thalmann, and Y. He, "Low-latency compression of mocap data using learned spatial decorrelation transform," *Comput. Aided Geom. Des.*, vol. 43, no. C, pp. 211–225, Mar. 2016.

[16] A. Sandryhaila and J. M. F. Moura, "Discrete signal processing on graphs," *IEEE Transactions on Signal Processing*, vol. 61, no. 7, pp. 1644–1656, April 2013.

[17] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," *IEEE Signal Process. Mag.*, vol. 30, no. 3, pp. 83–98, May 2013.

[18] S. K. Narang and A. Ortega, "Perfect reconstruction two-channel wavelet filter banks for graph structured data," *IEEE Transactions on Signal Processing*, vol. 60, no. 6, pp. 2786–2799, June 2012.

[19] S. K. Narang and A. Ortega, "Compact support biorthogonal wavelet filterbanks for arbitrary undirected

graphs," *IEEE Transactions on Signal Processing*, vol. 61, no. 19, pp. 4673–4685, Oct 2013.

[20] H. Q. Nguyen, P. A. Chou, and Y. Chen, "Compression of human body sequences using graph wavelet filter banks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 6152–6156.

[21] A. Anis, P. A. Chou, and A. Ortega, "Compression of dynamic 3d point clouds using subdivisional meshes and graph wavelet transforms," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 6360–6364.

[22] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249 – 270, 1980.

[23] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 129 – 147, 1982.

[24] C. Loop, C. Zhang, and Z. Zhang, "Real-time high-resolution sparse voxelization with application to image-based modeling," in *Proc. of the 5th High-Performance Graphics Conference*, New York, NY, USA, 2013, pp. 73–79.

[25] H. P. Moravec, "Sensor fusion in certainty grids for mobile robots," *AI Magazine*, vol. 9, no. 2, pp. 61–74, 1988.

[26] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *IEEE Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[27] K. Pathak, A. Birk, J. Poppinga, and S. Schwertfeger, "3d forward sensor modeling and application to occupancy grid based sensor fusion," in *Proc. IEEE/RSJ Int'l Conf. Intelligent Robots and Systems (IROS)*, Oct. 2007.

[28] R. Schnabel and R. Klein, "Octree-based point-cloud compression," in *Eurographics Symp. on Point-Based Graphics*, July 2006.

[29] Y. Huang, J. Peng, C. C. J. Kuo, and M. Gopi, "A generic scheme for progressive point cloud coding.," *IEEE Trans. Vis. Comput. Graph.*, vol. 14, no. 2, pp. 440–453, 2008.

[30] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach, "Real-time compression of point cloud streams," in *IEEE Int. Conference on Robotics and Automation*, Minnesota, USA, May 2012.

[31] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (PCL)," in *In Robotics and Automation (ICRA), 2011 IEEE International Conference on*. pp. 1–4, IEEE.

[32] C. Zhang, D. Florêncio, and C. Loop, "Point cloud attribute compression with graph transform," in *2014 IEEE International Conference on Image Processing (ICIP)*, Oct 2014, pp. 2066–2070.

[33] R. L. de Queiroz and P. A. Chou, "Compression of 3d point clouds using a region-adaptive hierarchical transform," *IEEE Transactions on Image Processing*, vol. 25, no. 8, pp. 3947–3956, Aug 2016.

[34] R. A. Cohen, D. Tian, and A. Vetro, "Attribute compression for sparse point clouds using graph transforms," in *2016 IEEE International Conference on Image Processing (ICIP)*, Sept 2016, pp. 1374–1378.

[35] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann, "Geometry and Attribute Compression for Voxel Scenes," *Eurographics Computer Graphics Forum*, 2016.

[36] R. L. de Queiroz and P. A. Chou, "Transform coding for point clouds using a Gaussian process model," *IEEE Trans. Image Processing*, 2016, submitted.

[37] J. Hou, L.-P. Chau, Y. He, and P. A. Chou, "Sparse representation for colors of 3d point cloud via virtual adaptive sampling," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, submitted.

[38] D. Thanou, P. A. Chou, and P. Frossard, "Graph-based motion estimation and compensation for dynamic 3d point cloud compression," in *Image Processing (ICIP), 2015 IEEE International Conference on*, Sept 2015, pp. 3235–3239.

[39] D. Thanou, P. A. Chou, and P. Frossard, "Graph-based compression of dynamic 3d point cloud sequences," *IEEE Transactions on Image Processing*, vol. 25, no. 4, pp. 1765–1778, April 2016.

[40] C. Zhang, D. Florêncio, and C. Loop, "Point cloud attribute compression with graph transform," in *2014 IEEE International Conference on Image Processing (ICIP)*, Oct 2014, pp. 2066–2070.

[41] R. L. de Queiroz and P. A. Chou, "Motion-compensated compression of dynamic voxelized point clouds," *IEEE Trans. Image Processing*, 2016, submitted.

[42] R. Mekuria, K. Blom, and P. Cesar, "Design, implementation and evaluation of a point cloud codec for tele-immersive video," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. PP, no. 99, pp. 1–1, 2016.

[43] R. Mekuria, Z. Li, C. Tulvan, and P. Chou, "Evaluation criteria for pcc (point cloud compression)," output

document n16332, ISO/IEC JTC1/SC29/WG11 MPEG, May 2016.

[44] H. S. Malvar, "Adaptive run-length/golomb-rice encoding of quantized generalized gaussian sources with unknown statistics," in *Data Compression Conference (DCC'06)*, March 2006, pp. 23–32.

[45] G. M Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," Technical report, IBM, Ottawa, Canada, 1966.