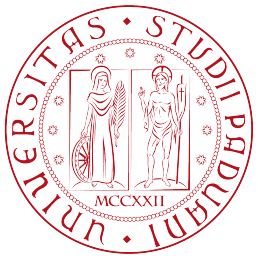


University of Padova
Department of Information Engineering
Master's Degree in Telecommunication Engineering

Color coding for 3D models in augmented reality applications

Supervisor
Prof. Simone Milani

Candidate
Enrico Polo
ID: 1124766

Academic Year: 2016/2017

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

In the last years, augmented reality (AR) has become a very attractive technological field, thanks also to the increasing diffusion of personal mobile devices capable to reproduce AR contents. Most AR applications deals with 3D visual data, i.e. 3D geometry and attributes such as color or normals. Such contents, which are usually characterized by a large amount of data, need to be efficiently compressed to be transmitted on different communication networks and displayed in heterogeneous AR applications. A diffused strategy to characterize 3D data at limited computational cost is the octree, which proved to be very useful in representing 3D raw data (point clouds) coming from sensors with a hierarchically-organized structure. In this thesis, we addressed the problem of compressing color information for voxelized point clouds using an underlying octree representation. We tested two different implementations: the RAHT coder and a more efficient implementation in computational therms. Moreover, we propose a low complexity prediction scheme to exploit temporal redundancies between consecutive frames in 3D video sequences. We showed that, with our prediction scheme we were able to save on average about 50% of the bits required without prediction.

1

Introduction

In this section we want to introduce the concept of augmented reality (AR) and to give a brief overview of its possible applications. Finally, we will focus on 3D image and video representation, pointing out the major issues regarding the transmission and the visualization of these data in AR applications.

1.1 Augmented reality overview

1.1.1 Definition

In general, the term *Augmented Reality* (AR) refers to human perception enhancement obtained by adding sensorial inputs, usually computer-generated, to the ones coming from the real world in order to provide additional information that is not available in the real scene.
Very often, AR is confused with *Virtual Reality* (VR) in which, instead, the user is isolated from the real world around him and completely immersed in a synthetic virtual environment.
For the sake of clarity and completeness, we report a schematic representation (Fig. 1.1) of the *Mixed Reality continuum* defined by Milgram in [7].

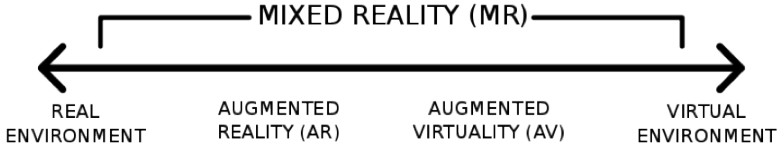


Figure 1.1.: Milgram reality-virtuality continuum.

As can be seen, VR and the real world are at the two extremes (i.e. in real environments there is no virtual content and vice versa), while AR and *Augmented Virtuality* (AV) are both in the middle (i.e. both are made of real and virtual elements). The difference between these two last categories is that in AR we add virtual signals (e.g. a 3D object or a sound source located somewhere in the space) into a real environment; in AV we add physical elements (e.g. a person or an object) into a virtual environment. From these premises, it is clear that AR systems supplement reality rather than replacing it, either completely (VR) or partially (AV). Note also that, all five senses can be improved with AR. Nevertheless, the main efforts and interests are focused on visual augmented reality, as it is the work reported in this thesis.
In order to summarize, it is possible to characterize an AR application as a system that

have all these characteristics [8]:

1) combines real and virtual objects;

2) allows a real time interaction between objects and user;

3) objects, scene and user point of views are registered with respect to a 3D reference system.

1.1.2 Applications

In the near future, the adoption of AR systems will be a groundbreaking innovation in many fields. In this section we will list some of them, providing some examples that already exists or will be available in the future [8], [9] and [10].

- *Medical*: AR systems can be used by doctors in surgery, e.g. for visualization tasks (especially in minimally-invasive operations where one can use imaging techniques, such as ultrasound or MRI, to obtain a real time volumetric rendering of the patient inside), for collaborative surgery and for training. Finally, it can be used to map informations or instructions directly on the patient.

- *Military*: also in this case AR can be used for training. It can be used also to display useful informations on the battlefield, to improve the skills of the soldiers (for example in case of low visibility).

- *Robotics*: remote manipulations (one can operate remotely with a virtual copy of the robot before sending the instructions to the real actuator, mitigating problems due to transmission delay or pilot-induced oscillations), human-robot interactions (robots can use AR techniques for communicating informations to humans or, vice versa, a human can use it to visualize inputs, outputs and states of the robot).

- *Manufacturing and Repairing*: as for surgery, AR can guide a technician in the assembly, repair or maintenance of complex machinery.

- *Entertainment and Gaming*: during the last years several games for mobile applications were commercialized online (e.g. Ingress). The application is set in the real world and the user, through the smartphone, sees virtual objects to interact with. Other possible applications are film production (to cut production costs), but also theatre (to add special effect to a live performance).

- *Marketing*: one of the hottest topics in this field is linked with e-commerce. In fact with the help of AR technologies, one can try clothes, shoes and so on before purchase them or see if a new piece of furniture fits well in a certain room, for example.

- *Tourism*: AR technology could provide historical informations about monuments or buildings around a city, or artworks inside museums.

- *3D teleconferencing*: an example is provided by the new system developed by Microsoft for the Hololens device called Holoportation [11]. Two remote people

wearing AR devices, can interact to each other as they physically are in the same environment (at least from an audio-visual point of view).

This last field is the one that inspired this work.

1.1.3 Systems

The main building blocks of an AR system, according to [12] are:

1) tracking and registration;

2) display technology;

3) real-time rendering.

The first block must provide a correct alignment between virtual objects and real environment. Furthermore, it must take care of the accurate registration of the 3D data to be displayed. In fact, high accuracy is required by AR systems to make the user perceive virtual and real objects seamlessly blended. Typically, hybrid tracking techniques are used by AR systems in order to accurately estimate sensor position and orientation, as well as to improve tracking robustness in different situations. Both position sensors (like for example GPS, accelerometer etc..) and computer vision techniques (which can be further divided into feature-based and model-based) are combined to the estimate the AR device pose with respect to the real world.

The second block, instead, refers to display technologies that make possible the contemporary visualization of both virtual and real objects. They can be grouped into three main categories [13]:

- *head mounted devices* (HMDs): they can be subdivided into *optical see-through* (OST) displays (Fig. 4.3a) and *video see-through* (VST) ones (Fig. 2.5b).

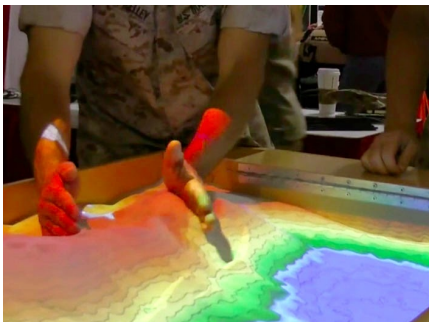
(a) OST: Microsoft Hololens. **(b)** VST: Samsung Gear VR.

Figure 1.2.: HMD devices examples.

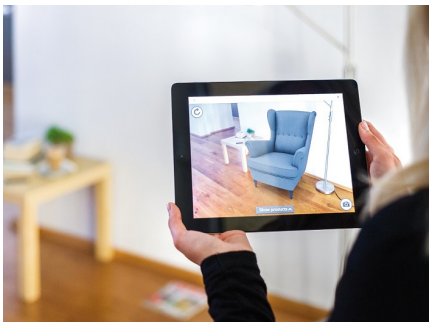
The former technology offer an instantaneous and natural view of the real world, using a transparent screen to superimpose virtual contents. The latter displays virtual and re-acquired real objects on a standard video screen and ensures a better consistency between virtual and real representation on the display.

- *Projection-Based Devices*: conversely from HMDs, with this technology there is no need to wear any invasive device to visualize virtual contents. In fact, virtual contents are directly projected into objects in the real world (Fig. 1.3a). The main drawback of such a system is the low portability.

- *Handheld Devices*: this technology is probably the one that offers the worse immersive experience but, on the other side, actually is the best for mobility, costs and availability. In fact, it includes smartphones, tablets and all kind of mobile devices (Fig. 1.3b).



(a) Projection-based display: sandtable for military planning.



(b) Handheld display for AR: tablet.

Figure 1.3.

Finally, the last block is the one responsible for 3D data storage, transmission (if required by the application) and rendering in a fast and realistic way, in order to enable interactivity with virtual objects. Our work is focused on this research area and our software is suitable for running on HMDs, can be extended to deal with handheld devices, while projection-based technologies are not properly our target.

1.2 3D models representation

Until now, both static and time-varying 3D objects, along with their associated color information, have been represented using *polygonal meshes* and *point clouds*[6]. For this reason, the following section will briefly describe them, pointing out their strengths and weaknesses.

1.2.1 Point cloud

Generally speaking, a point cloud is a set of 3D data points in some reference system. In our case, we can assume without loss of generality the $(\hat{x}, \hat{y}, \hat{z})$ three-dimensional basis as reference system. A 3D virtual object is represented through a subset of points belonging to its surface, obtaining a salt-n-pepper representation like in Fig. 1.4. Of course, each point can also have other attributes besides coordinates, like color.

Figure 1.4.: Cup point cloud.

As we can see, it appears like a sampled version of the object's surface. For this reason, point cloud is very suitable for representing raw data coming directly from 3D acquisition sensors (e.g. laser scanner, ToF devices etc..).
On the other side, this approach is not very good for a good rendering. Indeed, point clouds do not provide any additional information about the connectivity among points (i.e. which points belong to the same surface), texture maps and so on. Hence, in order to obtain good visualization results we need further processing (as we will see in the next subsection) or a very dense representation. Unfortunately, the denser the model, the higher the number of bits required to represent it. Furthermore, a lot of computation and more expensive sensors are required to handle huge amounts of 3D points.

1.2.2 Polygonal mesh

A polygonal mesh is defined as a pair $M = (V, F)$ where V is a list of vertices (points in the 3D space similarly to point clouds) and F is an abstract simplicial complex that represents connectivity information between vertices (the mesh topology).
In practice, F is a simplicial complex of order two, it means that its constitutive elements (simplexes in general) can only be vertices, edges or faces. So, in an informal but more intuitive way, we can define a mesh as a collection of vertices, edges and faces, connected in order to compose a polyhedron that approximates the shape of a 3D object. Typically, faces are triangles (triangular meshes), but there are also some implementations with quadrilaterals or (rarely) general convex polygons. In Fig. 1.5 and 1.6, we can see the difference between point cloud and triangular mesh representation of the same object.

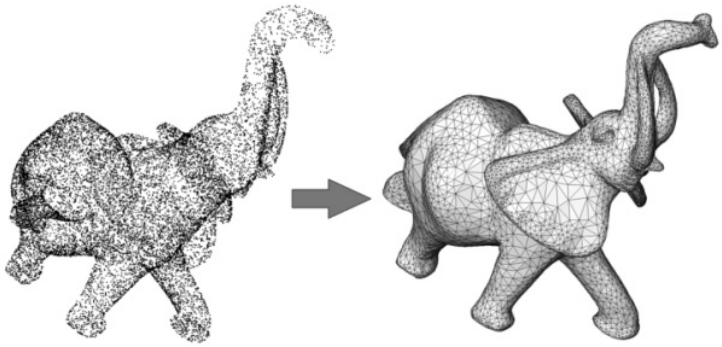


Figure 1.5.: Point cloud vs triangle mesh comparison: elephant.

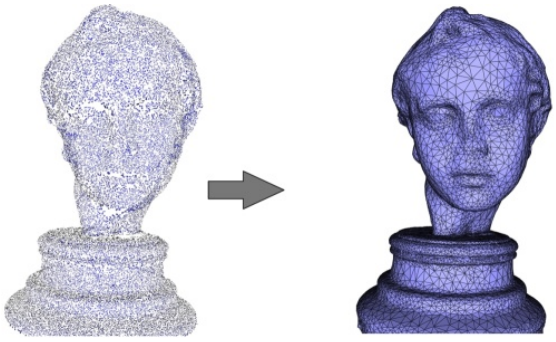


Figure 1.6.: Point cloud vs triangle mesh comparison: statue.

According to Figure 1.5 and 1.6, it is evident that for visualization purposes mesh representation is much preferable than point cloud (even if no rendering technique is applied in both models). In fact, this approach is widely used in computer graphics where there are no real time constraints between data acquisition and model generation (i.e. it can be done offline). Mesh representation, in fact, requires a lot of computational demanding preprocessing operations that must be done to generate good meshes from an acquisition (e.g. denoising, surface reconstruction etc..).

1.3 Problem statement

As already mentioned, our work is focused on AR applications that require real time data acquisition and transmission or storage, like immersive communication. In all the applications where the content to be displayed can be generated offline and stored in the AR device (i.e. it does not have to be transmitted), the processing time and complexity constraints are less tight and more traditional strategies can be adopted.
In the considered case, the challenge is to develop an algorithm that provides:

1) *High compression*: 3D data are, by their nature, very large in size. For this reason we need to compress both geometry information and attributes (e.g. color), not only to efficiently store them, but also to minimize as much as possible the final bit rate to be transmitted, in order to satisfy real time constraint.

2) *Good quality rendering*: compressed 3D data should be organized in a way that allows the decoder to visualize them quickly, ensuring both a high visual quality and a seamless interaction for the user, without the need of heavy further computation.

3) *Real time execution*: of course the algorithm itself must perform compression and decompression very efficiently, otherwise the benefits of good compression for reducing the transmission time would be lost. Regarding storage, there is also the need for real time execution in order to code and decode 3D sequences with good frame rates.

Remembering what reported in section 1.2, real-time AR applications prefer point clouds to avoid preprocessing overhead and, as we will see in the next chapter, to achieve a high compression ratio with a limited computational effort.

Conversely, we should prefer meshes to achieve good quality rendering and to improve compression in dynamic scenes, exploiting temporal consistency. Unfortunately, point cloud to mesh conversion during a live capture and the generation of temporal consistent meshes seems to be a prohibitive task from a computational point of view [6]. For this reason, researchers are still trying to find a solution that can satisfy requirements 1),2) and 3) at the same time. The strategies designed so far aim at representing point cloud raw data in a compact way which is also suited for rendering, using simpler structures than meshes.
The work reported in this thesis is focused only on the compression of voxel attributes; more precisely, the proposed scheme deals with the compression of color components.

2

Real time 3D data compression

In this chapter we want to present some state-of-the-art solutions in real time 3D data compression, reporting some promising approaches developed by researchers for both geometry and color compression.

2.1 Geometry

Nowadays, the most efficient way to compactly represent point cloud spatial distribution is using *voxels*. Moreover, voxel data can be rendered quite easily and quickly, similarly to meshes. A voxel is a sample on a regularly spaced $N \times N \times N$ three-dimensional grid. It can be thought as the 3D equivalent of a pixel. In Fig. 2.1, we report a simple example of a voxel grid with $N = 4$, where a voxel is highlighted in green. Each voxel can be associated to different types of information. When representing the geometry of a 3D model, voxels correspond to binary variables that are set to 1 whenever part of the object lies within its volume; in case the voxel is empty, the corresponding variable is set to 0.
When referred to a point cloud model, a voxel is set to 1 whenever one or more 3D points are lying within its boundaries.

It is also possible to associate voxels to color information. In case a voxel is non-empty (i.e. it contains part of the 3D volume), it is possible to associate three color components R, G, B to it. In the following, we will focus on the characterization of object volumes, i.e., each voxel will be characterized by a binary value.

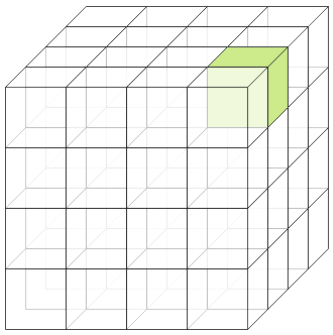


Figure 2.1.: $4 \times 4 \times 4$ voxel grid.

Such a voxel grid can be, in turn, efficiently stored using an *octree* data structure, which is a tree where each internal node must have exactly eight child nodes.
The octree's root represents the entire $N \times N \times N$ voxel volume V, while each of its

child nodes represents an $\frac{N}{2} \times \frac{N}{2} \times \frac{N}{2}$ different sub-volume V_i, such that:

$$\bigcup_{i=1}^{8} V_i = V \tag{2.1}$$

$$V_i \cap V_j = \emptyset, \forall i \neq j \tag{2.2}$$

Each child node can assume a binary value: if set to 1 the resulting $\frac{N}{2} \times \frac{N}{2} \times \frac{N}{2}$ sub-volume has to be considered occupied (i.e. it contains at least a voxel that is part of the 3D object); otherwise, the corresponding sub-volume is empty (i.e. it's not occupied by the object we want to represent). This spatial decomposition is iterated on each node set to 1, until each sub-volume has dimension $1 \times 1 \times 1$ (i.e. they have reached the voxel's dimension). Empty nodes, instead, don't need further iterations because they represent an empty region and the corresponding sub-tree nodes will be set to 0. In Fig. 2.2 we outline this procedure to encode a simple 3D patch within a grid with $N = 4$ using an octree. The resulting volumetric representation is usually referred as *Sparse Voxel Octree* (SVO) data structure.

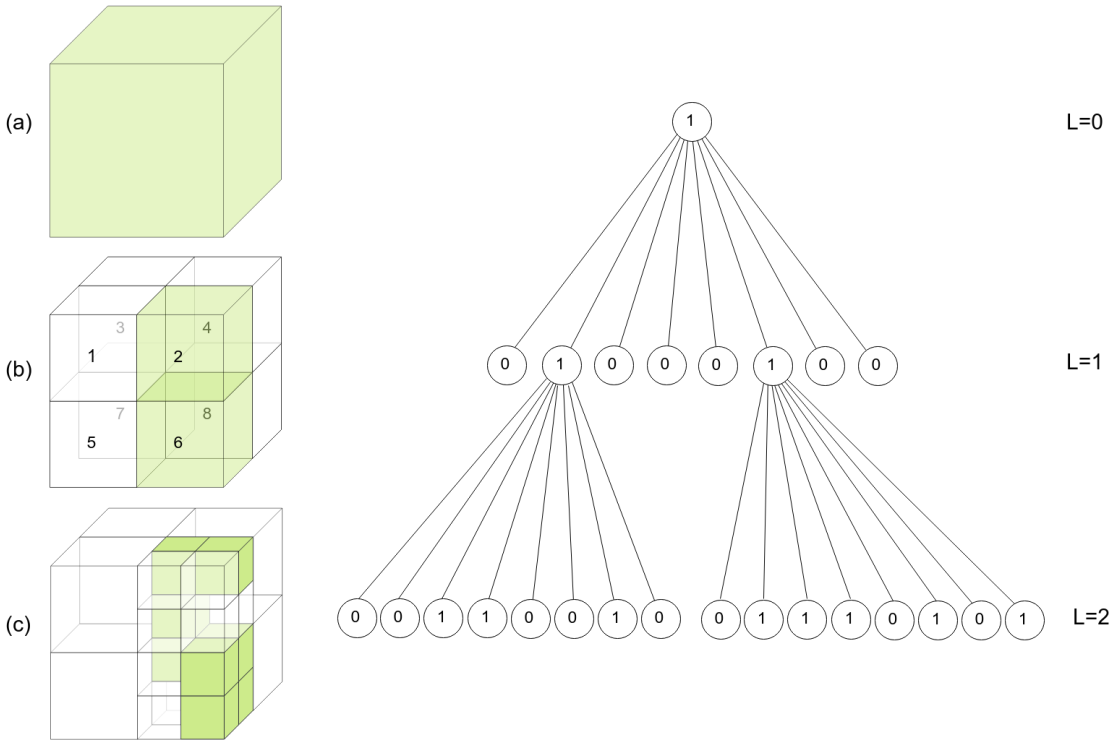


Figure 2.2.: $4 \times 4 \times 4$ voxel grid and its octree representation. 3D patch representation at **(a)** $1 \times 1 \times 1$, **(b)** $2 \times 2 \times 2$ and **(c)** $4 \times 4 \times 4$ resolution.

In this example, we have assumed that each partition of a certain volume is scanned in a raster scan order, as indicated on the grid corresponding to the first decomposition ($L = 1$). It allows us to make few observations about SVOs, before proceeding explaining how they are used to compress point clouds:

- Octree representation avoids wasting a lot of bits to code empty regions. In this example the resulting binary sequence is 25 bits long (24 bits if we consider the

fact that root usually is not coded and assumed always set to 1). In fact, if one coded directly the $4 \times 4 \times 4$ voxel grid using the same binary representation (1 for occupied voxels, 0 otherwise), it would need 64 bits. One can notice that, in the worst case (i.e. all the voxels occupied), with octree we'd need $1+8+8 \cdot 8 = 73$ bits to encode this simple grid but, in practice, this is not a likely situation. Indeed, in our target applications, occupied voxels within a grid are typically $\leq 1\%$ of their total number (and in case of very dense grids the percentage does not exceed 2% or 3%). This explains why the term *sparse* is used to define this data structure.

- Each decomposition level L, which is also called Level of Definition or LoD, can be considered as a refined representation of the 3D object within the voxel grid. In particular, starting from the octree's leaves and climbing level by level until the root, we decrease the spatial resolution of the correspondent grid by a factor 2 in all the tree dimensions.

SVOs were developed to represent 3D geometry for the first time in the 1980s in [14] and then in [15]. The first that used SVOs in order to compress point cloud data was [16]. The process consists basically of three steps:

1) *Voxelization*: the point cloud $\mathcal{P}$ is enclosed in a bounding cube $\mathcal{V}$ and its spatial distribution is quantized. The quantization process consists in dividing $\mathcal{V}$ into $N \times N \times N$ voxels. Typical values of N are 128, 256 or 512 (which respectively correspond to $L = 7$, 8 or 9). If a voxel contains at least one point of the original point cloud it's set to 1, otherwise to 0.

2) *Octree computation*: once the point cloud has been voxelized, the corresponding octree representation is generated, as just illustrated, building an SVO structure.

3) *Octree encoding*: in order to further improve compression, one can use transformations or *intra* frame predictive schemes instead of the simple binary representation described in Fig. 2.2; in order to reduce the redundancy in 3D voxel structures.

More precisely, the last step is the entropy coding of the data after 1),2) and 3). Although the entropy coder should be carefully chosen and designed in order to obtain the best possible results (in therms of both compression and execution time), in this work we focus on the signal processing steps that permits maximizing the coding performance.
It is also worth underlying the intrinsic concurrency of tasks that this scheme provides. There's no need to wait operation 2) to be done for the full octree, before performing 3); moreover, operations can be done level by level to increase the efficiency.
Another challenging task concerns the compression of dynamic sequences. In this case one can perform some kind of *inter* frame 3D prediction scheme (like it is done in all the actual 2D video codecs); in order to reduce the number of voxel to be coded, thanks to the temporal redundancy.
In the following, we will report some of the most interesting works (from our point of view) for either static frames and dynamic sequences encoding.

2.1.1 J. Kammerl et al. [1]

In this paper authors propose a generalized and extensible point cloud compression system that can be tuned to meet different resolution, complexity and accuracy requirements. Such scheme can be easily adapted to the compression of voxel side attributes (like, for example, color or point cloud residuals). Anyway, the approach was mainly focused on keeping low computational complexity and on providing a fast implementation of the compression algorithm. In fact, their source code and relative documentation was part of the Point Cloud Library (PCL) [17].
This coding scheme was intended for robotics applications like *teleoperation* and *remote data processing* Fig. 2.3, but it can be extended to many other scenarios (i.e. that must satisfy the constraints illustrated in section 1.3).

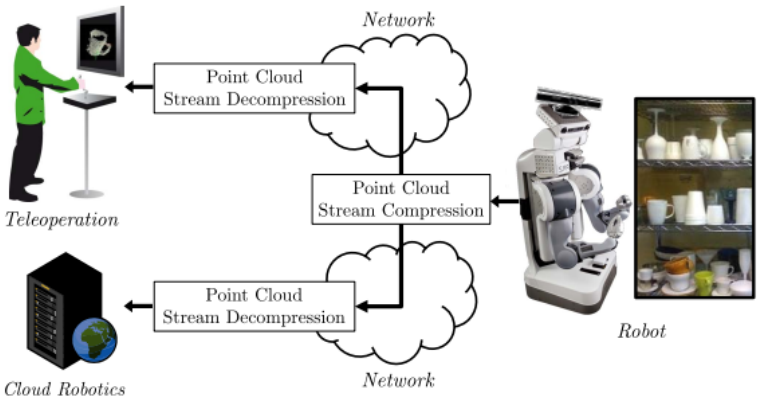


Figure 2.3.: Schematic overview of compressed point cloud streaming scenarios. [1]

Given a static input frame (i.e., in this case, a point cloud which reproduce the 3D model of the object at a given time instant), an octree hierarchical decomposition of the occupied space is adopted, building a coding tree in depth first order. The decomposition stops either when a certain LoD $\bar{L}$ is reached or when the number of 3D points within the underlying subvolume is lower than a threshold. In both cases the stop condition is determined a priori. Instead, when serializing the octree the traversal is done in a breadth first order starting from the root, so that one can progressively encode and decode the stream. With this method 8 bits for each branch node is required to encode geometry.
Regarding dynamic sequences, they propose a method to remove temporal correlation between consecutive SVO representations of the acquired point clouds. In fact, direct correspondences between adjacent point clouds are really challenging to be detected. For this reason they proposed a novel double-buffering octree scheme in order to enable comparison and differential coding of consecutive SVOs.
When coding the first frame, of course, no previous frame is available and the coding procedure is the one already discussed. The only difference is that, instead of storing the 8 child nodes (also known as *octants*) for each branch node using a single byte, they store them using an additional one, which refers to the previously coded frame. Note that for the first frame is always set to 00000000. Let us call the subset of bytes associated to the current frame A and the subset of bytes associated to the current frame B.

When the next point cloud needs to be coded, the correspondent SVO will be saved in B. In this way, a copy of the SVO generated for the first frame is stored in A and, in parallel, the SVO that represents the actual frame is stored in B. In general, it is not guaranteed that the second structure will fit into B. For this reason while building the octree of the actual frame we have two cases:

- octants of a certain branch node exist in A: the correspondent byte in B is filled accordingly;
- octants of a certain branch node doesn't exist in A: two new bytes are allocated for representing them, one belongs to A and is set to zero, the other is in B and represents the octants configuration.

Once done, the octree B is traversed in a breadth first order to be serialized as when coding a static frame. Instead of encoding the byte in B a XOR operation is done between the actual byte and the correspondent in A. In other words, we encode only the difference between actual and previous frame, level by level. A schematic representation of this operation is reported in Fig. 2.4

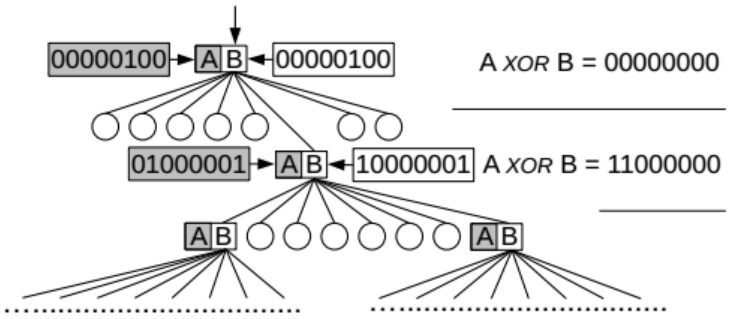


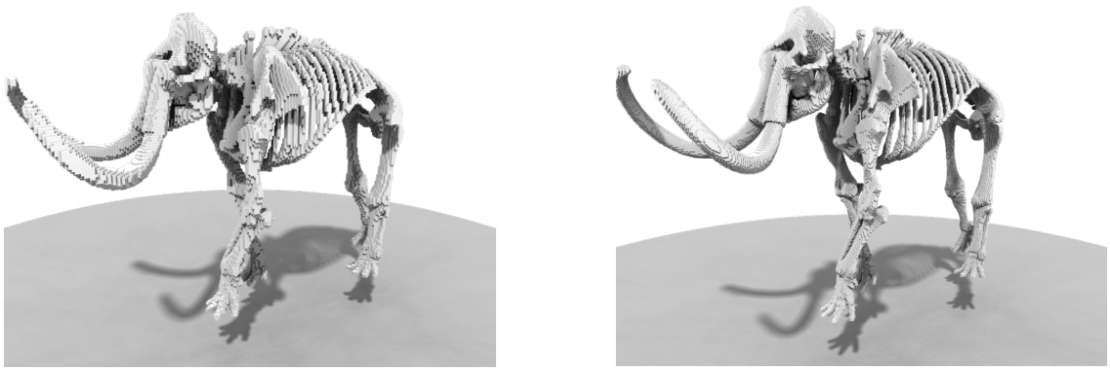
Figure 2.4.: Differential encoding of a consecutive point cloud using buffer B. Buffer A contains the previously processed octree structure.[1]

When coding the subsequent frame, A will store the previous octree while the actual one will be stored in B, and the process is repeated switching the roles of A and B until the sequence is over or in case one wants to add a new reference frame coded independently. In this case the double-buffering structure is resetted and the procedure restarts. Optionally, another step can be performed: the coding of the original point cloud attributes within each voxel (e.g. the point residuals with respect to the voxel origin) can be performed, increasing the reconstruction accuracy, but reducing the compression ratio. Nevertheless, our approach is focused on the compression of voxels.
The final stage of the compression routine for a frame is the entropy coding of octree bytes, which is performed using an integer approximation of the *arithmetic coder*. Prior to every entropy encoded bit stream, corresponding symbol frequency tables are additionally encoded.
We want to conclude reporting the results obtained by the authors in their tests. They notice that for high target precision (i.e. high values of L, the maximum decomposition level), using differential coding instead of the static one provides a gain of about 34% in terms of compression ratio. This improvement is less effective when the precision

decreases, because the fixed size of the frequency table encoding dominates the coded stream as the compressed data size decreases, leading to a saturation effect.

2.1.2 S. Milani [2]

In this paper the author focused on increasing the coding efficiency of the SVO representation, by applying a so called *Cellular Automata* (CA) reversible transform on hierarchical decomposed voxel volumes. Moreover, the flexibility and the quality of the visual rendering can be improved with respect to the one obtained using SVO (Fig. 2.5).



(a) SVO. **(b)** CA.

Figure 2.5.: SVO and CA transform based model reconstruction at equal rates (120Kbit/s). [2]

It can be seen as a transform operated on each octant of a SVO, starting from its leaves and proceeding in breadth first order following a bottom up fashion up to the root. Nevertheless, there is no need to explicitly build the octree and then transform it: the transformation can be done directly while scanning the voxel volume.
The $N \times N \times N$ grid C^N is divided into non-overlapping $2 \times 2 \times 2$ sub-volumes (blocks). Within each block $\mathbf{b}$ voxels (which can be associated to a cellular automaton s_i whose state can be 0 or 1 depending on whether they are occupied or not) can assume $2^8 = 256$ different possible configurations. These can be mapped to integer numbers in the interval $[0, 255]$. CA-transform operates an invertible remapping of the vectors $\mathbf{b}$ into $\mathbf{b}^P = P(\mathbf{b})$ which allows a more efficient data organization. After transformation has been performed in all the sub-volumes, corner automata are grouped together in a volume of halved dimensions $C^{\frac{N}{2}}$ in order to be further processed (in fact they represent the DC component of the transform). The remaining coefficients instead are entropy coded for the transmission (they represent the AC component of the transform). The process can be iterated on the DC subvolume a certain number of times until no additional decomposition can be applied; Fig. 2.6 reports an example.

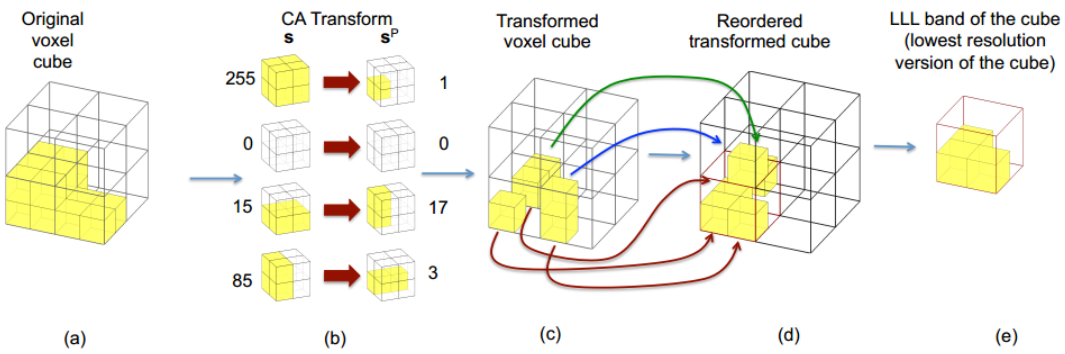


Figure 2.6.: Example of Cellular Automata block transformation of a $4 \times 4 \times 4$ voxel block. [2]

In order to obtain effective improvements with respect to SVO representation the transformation P must be designed properly. Defining $n_0(\mathbf{b})$ as the number voxels within a certain block equal to 0 and $n_1(\mathbf{b})$ the number of voxels equal to one a good transformation should increase the quantity $n_0(\mathbf{b}^P)$, so that, the number of AC bands to be coded will decrease. In order to do this, the author designed $P(\cdot)$ such that:

1) $P(0) = 0$;

2) $P(255) = 1$;

3) high probable blocks $\mathbf{b}$ with $n_1(\mathbf{b}) > 4$ should be converted to $\mathbf{b}^P$ such that $n_0(\mathbf{b}^P) = n_1(\mathbf{b})$;

4) less probable blocks $\mathbf{b}$ with $n_0(\mathbf{b}) > 4$ should be converted to $\mathbf{b}^P$ such that $n_1(\mathbf{b}^P) = n_0(\mathbf{b})$;

5) high probable blocks $\mathbf{b}$ with $n_0(\mathbf{b}) > 4$ should be converted to $\mathbf{b}^P$ such that $n_0(\mathbf{b}^P) = n_0(\mathbf{b})$;

6) high probable blocks $\mathbf{b}$ should have $s_i^P = 1$ for small i values.

These properties can be used in the design stage starting from block probability distribution for the considered 3D model. In the paper implementation the transform is designed offline and the probability distribution is estimated using a training set of three 3D models. In case of video sequences, one can code the first frame using the statistic estimated from the training set. For the next ones one can update $P(\cdot)$ according to the probability distribution estimated from previous frames, further improving transform performances.
All AC coefficients and the remaining DC (the one resulting from the last iteration that need to be sent) are coded using an *adaptive arithmetic coder*.
To provide a fair comparison between CA transform and standard octree coding the author has tested both procedures on a bunch of static models using the same entropy coder as final step. Such a test on voxel grids with $N = 512$ showed that on average the implemented CA transform permits reducing the bit stream of 34% with respect to the standard strategy.
About the computational complexity, it increases of about 90% with respect to the standard strategy, although the algorithm still runs in real time.

2.1.3 D. Garcia and R.L. de Queiroz [3]

In this paper, similarly to the first one presented, the authors try to improve compression in dynamic sequences using an inter frame prediction scheme, as it is done for 2D video. The main difference between 2D video and 3D ones is that in the first case the amount of pixels is the same along frames, while in the second one the number of points in subsequent point clouds (or the number of occupied voxels in adjacent frames) changes a lot. Consequently, the length and configuration of consecutive octrees can be really different: nodes that were not filled in a previous frame could suddenly be occupied in a new frame, or vice-versa (as we already saw in subsection 2.1.1).
Given this, the goal of the authors is to build a good prediction octree $\mathbf{O}_p$, starting from a reference frame $\mathbf{O}_r$, that has as many bytes as the current frame we want to predict $\mathbf{O}_c$ (i.e. it has the same octree structure).
In order to do this, they proceed copying from $\mathbf{O}_r$ to $\mathbf{O}_p$ only those bytes (i.e. octants) that are present also in $\mathbf{O}_c$. Then, for all the bytes that are present in $\mathbf{O}_c$ but not in $\mathbf{O}_r$, $\mathbf{O}_p$ is filled with zeros, with a procedure similar to [1]. In this way $\mathbf{O}_p$ has the same bytes as $\mathbf{O}_c$ and $\mathbf{O}_r$ is used for predicting only common branches. The difference with respect to the previous work is that instead of making $XOR(\mathbf{O}_p,\mathbf{O}_c)$ and entropy coding, here $\mathbf{O}_p$ is used to sort $\mathbf{O}_c$ before entropy coding. In fact, $\mathbf{O}_p$ is sorted in ascending order and the same sorting order found is applied to $\mathbf{O}_c$ obtaining $\mathbf{O}_{cs}$ that will be entropy coded.
In this way, in case of good prediction $\mathbf{O}_{sc}$ will contain long sequences of bytes encoding the same value, improving entropy compression performances. Conversely, if the prediction is not good the compression results will be similar to the ones obtained when static coding $\mathbf{O}_c$.
Moreover, authors in [3] extend this method allowing the use of more than one reference frame for the prediction: in this case $\mathbf{O}_r$ will be composed by the union of bytes representing 3D points in each reference frame.
In order to correctly decode frames, the decoder needs to restore the original bytes' order. For this reason, the encoder must provide also the transformation performed when sorting $\mathbf{O}_p$, that is applied to $\mathbf{O}_c$. So, an histogram $\mathbf{H}_r$ is computed for the reference frame, and the same in done for $\mathbf{O}_{ps}$, the sorted version of $\mathbf{O}_p$ obtaining $\mathbf{H}_{ps}$. Finally, the difference $\mathbf{H}_d = \mathbf{H}_r - \mathbf{H}_{ps}$ is entropy coded and sent using M-bit representation, with $M = \lceil log_2(argmax(\mathbf{H}_d)) \rceil$. The decoder using $\mathbf{H}_d$ to $\mathbf{H}_r$, computed while decoding the previous frame, can obtain $\mathbf{H}_{ps} = \mathbf{H}_r - \mathbf{H}_d$ and use it to reorder the decoded frame $\mathbf{O}_{cs}$ re-obtaining $\mathbf{O}_c$.
They tested three configurations of their algorithm with one, two or three reference frames for the prediction. They compare their procedure with simple octree coding followed by entropy coding and with [1]. The entropy coder used in all this cases was *deflate* (GZIP version 1.0.7 of the Keka archiver for OSX, with the maximum-compression setting -mx9). The dataset used was composed by seven sequences, each 200 frames long.
Regarding the three version of the algorithm they found that using only one reference frame, besides being more computational efficient, provide also the best compression performances. Moreover, they found that with respect to standard entropy coded octree, they gain 5% on average in the compression ratio; while with respect to [1] they

gain about the 8% on the tested sequences. Nevertheless, they notice that their method, but also the one in Kammerl et al., performs worse as the movement within the frame increases. For this reason they suggest the use of motion compensation strategies to improve performances.

2.2 Color

In this section we are going to present two different approaches that can be found in literature in order to efficiently compress the color attributes of a point cloud.
Even though, we present color and geometry coding as two separate processes, they are not really independent to each other, in the sense that, the basic strategy to compress color is very similar from the ones just presented for geometry. In fact the first step is once again the voxelization of the color attributes. In the case of color information, the voxel value does not represent whether it is occupied or not; each voxel is associated to color components, which are specified only for the occupied ones.
Assuming that colors in the point cloud are represented in an RGB color space we have that, without compression, each color is represented by a triplet, where each component takes values in a range $[0, 255]$. So, each 3D point requires $8 \cdot 3 = 24$ bits to represent its color attributes. During voxelization, whenever a voxel is not empty, a single triplet is used to represent its color attributes. Let us call p the number of points of the point cloud within a voxel. The RGB values associated to that voxel will be:

$$R = \left[\frac{1}{p}\sum_{i=1}^{p} R_i\right]; \qquad G = \left[\frac{1}{p}\sum_{i=1}^{p} G_i\right]; \qquad B = \left[\frac{1}{p}\sum_{i=1}^{p} B_i\right]; \tag{2.3}$$

where R_i, G_i, B_i are respectively the red, green and blue component of the i-th point within the voxel and $[\cdot]$ is the nearest integer function. In this way, one needs 24 bits per voxel thus reducing the size of the color representation according to:

$$C_{vox} = \frac{n_{occ}}{\sum_{j=1}^{n_{occ}} p_j} \cdot C_{pc} \tag{2.4}$$

where C_{vox} is the number of bits required for representing color after voxelization, C_{pc} are the bit required to code color in the point cloud, n_{occ} is the number of occupied voxel in the voxel grid and $p_j \geq 1$ is the number of points of the point cloud within the j-th occupied voxel. In this formulation, we assume that the voxelized geometry of the 3D model is available. In practice, we assume the voxelized volume is coded using an SVO approach. As a matter of fact, we can store color attributes using a simpler structure (i.e. an array), where the first attribute corresponds to the first occupied voxel in the last level of the octree, the second one to the second occupied voxel and so on, according to some scanning order.
Although this organization of color data requires only 24 bits per occupied voxel (bpv), the final bit stream size will be anyway significant (typical bit stream size to encode geometry are in the order of $1 \div 5$ bpv). Of course, entropy coding will reduce the bit rate required but its coding gain could not be enough.

For this reason, researchers are focused on finding transformations of the color attributes that, once again, should provide high compression at the lowest computation cost. Note that, in this case, one deals with triplets of integer data, not with binary values as before, so the computational complexity increases a lot. In the next subsections we will report two papers that propose two different solution for attribute coding.

2.2.1 C. Zang, D. Florêncio and C. Loop [4]

In this work authors proposed a method based on *Graph Transform* (GT) to compress point cloud attributes, in particular color and normals. Graph transform is a linear transform that assumes an underlying graph structure. In the mentioned case a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is built starting from a voxelized representation of the point cloud, where weights w_i are defined on edges $\mathcal{E}$ and shall represent attributes similarly between nodes (their values are inversely proportional to the distances between voxels). In this way one can build an adjacency matrix $\mathbf{A}$ representing the graph upon which the transform is applied. To make it clear, we report an example taken from the paper where this procedure is illustrated on a small $4 \times 4 \times 4$ octree (Fig. 2.7).

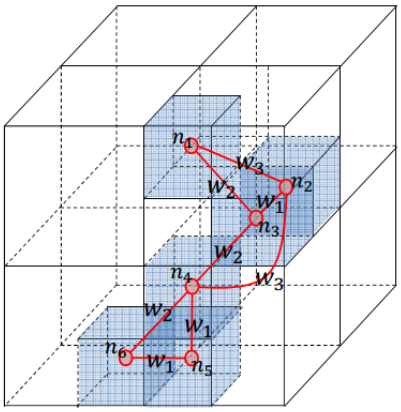


Figure 2.7.: Example of graph built upon a $4 \times 4 \times 4$ octree. [4]

The correspondent adjacency matrix is defined as:

$$\mathbf{A} = \begin{pmatrix} 0 & w_3 & w_2 & 0 & 0 & 0 \\ w_3 & 0 & w_1 & w_3 & 0 & 0 \\ w_2 & w_1 & 0 & w_2 & 0 & 0 \\ 0 & w_3 & w_2 & 0 & w_1 & w_2 \\ 0 & 0 & 0 & w_1 & 0 & w_1 \\ 0 & 0 & 0 & w_2 & w_1 & 0 \end{pmatrix} \tag{2.5}$$

Defining a multivariate Gaussian random vector $\mathbf{x} = (x_1, ..., x_n)$ on the graph nodes (in this example $n = 6$), it is possible to describe its probability density function as:

$$p(\mathbf{x}) = (2\pi)^{-\frac{n}{2}} |\mathbf{Q}|^{\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{Q}(\mathbf{x}-\mu)} \tag{2.6}$$

where μ is the mean of the distribution and $\mathbf{Q} \geq 0$ is the precision matrix (i.e. the

inverse of the covariance matrix $\mathbf{\Sigma}$). It can be shown that the eigenvector matrix of $\mathbf{Q}$ is equal to the Karhunen-Loève Transform (KLT), which is the optimal solution for decorrelating the input signal (i.e. for compressing it). So, once again referring to the example depicted in Fig. 2.7, it is possible to derive the precision matrix associated to $\mathcal{G}$ as:

$$\mathbf{Q} = \delta(\mathbf{D} - \mathbf{A}) \tag{2.7}$$

where δ is a constant dependent on the signal's variance and $\mathbf{D} = diag(d_1, ..., d_6)$ with $d_i = \sum_j a_{ij}$, the elements of $\mathbf{A}$.
Finally, performing eigenvalue decomposition of $\mathbf{Q}$ as:

$$\mathbf{Q} = \mathbf{\Phi}\mathbf{\Lambda}\mathbf{\Phi}^{-1} \tag{2.8}$$

where $\mathbf{\Lambda} = diag(\lambda_1, ..., \lambda_n)$ is the diagonal matrix containing the eigenvalues of $\mathbf{Q}$ and $\mathbf{\Phi}$ is the relative eigenvector matrix, one can find the transform it was looking for, that is, in fact, $\mathbf{\Phi}$.
Following this procedure authors described an algorithm to compress and encode attributes that can be summarized as follows:

1) Partition the voxel grid in i equal sized sub-volumes (blocks).

2) For each block build the correspondent graph and compute the $N_i \times N_i$ transform $\mathbf{\Phi}_i$, where N_i is the number of occupied voxels within the i-th block.

3) Perform the transform on the vector storing the attributes for that block $\mathbf{y_i}$ obtaining $\mathbf{f}_i = \mathbf{\Phi}_i\mathbf{y}_i$. If $\mathbf{y}_i$ stores color attributes, transformation and encoding will be done separately for each of its three components.

4) Quantize the transformed coefficient obtaining $\mathbf{f}_{qi} = [\mathbf{f}_i/q]$, where q is the quantization step.

5) Entropy code $\mathbf{f}_{qi}$ using an arithmetic coder that assumes a Laplacian distribution of the transformed coefficients.

Notice that, at point 4) quantization is performed so, the decoder will not be able to reconstruct original color values $\mathbf{y_i}$, i.e., it is a *lossy* coding scheme. For this reason, one cannot focus only on compression ratios, as it was for geometry coding, but has to take care also of the resulting distortion introduced by quantization. A useful tool in these situations are Distortion-Rate curves that reports the measured distortion while decoding data (w.r.t. the original ones) as a function of the rate needed to encode them. In our case, the rate is usually indicated as bpv (as it was for geometry) while for the distortion metric usually is adopted the PSNR defined as:

$$PSNR = 20\log_{10}\frac{max(c)}{\sqrt{MSE}} \qquad MSE = \frac{1}{N}\frac{1}{S}\sum_{j=1}^{N}\sum_{k=1}^{S}(y_{jk} - \hat{y}_{jk})^2 \tag{2.9}$$

where $max(c)$ is the max value a color component can assume (in case of 24-bit RGB color space it is 255), $N = \sum_i N_i$ is the total number of occupied voxels, S is the

number of coordinates needed to represent the attribute (in case of color $S = 3$), while $y_j = (y_{j_1}, ..., y_{j_S})$ and $\hat{y}_j = (\hat{y}_{j_1}, ..., \hat{y}_{j_S})$ are respectively the vectors representing the original j-th voxel's attribute and the corresponding reconstructed value.
The authors compared graph transform, with the N-points one dimensional DCT and with the color coder implemented in PCL [17] on 6 video sequences voxelized at a resolution of $512 \times 512 \times 512$ voxels. They showed that, GT outperforms both methods, in a Rate-Distortion sense. Respect to PCL coder, for example, for a reconstruction error of 38 dB, GT requires 0.36 bpv instead of 8.4 bpv required using PCL. With respect to DCT, using the same block sizes in both cases, GT improves the performances, in terms of PSNR, of about 1-3 dB, at the same bitrate; while, given a target PSNR GT requires in mean half the bits required using DCT. Moreover, they showed that the larger the block size, the better are compression performances. This is due to the fact that more voxels are decorrelated together in each block. On the other side, from a computational point of view, the eigenvalue decomposition necessary to compute $\mathbf{\Phi}_i$ is $O(n^3)$, thus increasing the block size will increase a lot the execution time. It appears clear when looking at the results they obtained using different block sizes. With their setup and implementation, GT using $2 \times 2 \times 2$ voxel blocks requires on average about 0.85s to be computed, obtaining PSNR = 38 dB at a rate of 0.4 bpv; while using 16x16x16 blocks it requires 109.8 minutes, but at the same rate PSNR increases a lot, being about 44 dB.

2.2.2 R.L. de Queiroz and P.A. Chou [5]

In this work, authors developed a region-adaptive orthogonal transform to encode color attributes in point clouds. Their method performs a bottom up scanning of the octree, until reaching the root. While performing the three traversal, color attributes associated to the leaves of a certain branch at a certain level are grouped together in the so called DC term, that contains color information representing the entire branch and will be further processed. The residuals of this grouping operation, instead, are entropy coded to be transmitted (or stored). Once a level $\bar{L}$ is transformed, the procedure is iterated in the next one $\bar{L} - 1$ between all leaf nodes at that level and the DC terms representing lower branches already transformed.
More in detail, in the first step the transform operates in all $2 \times 2 \times 2$ blocks that contains at least one occupied voxel. Within each block voxels are transformed at first along x dimension, then along y and finally along z. In this way, in an SVO with $\bar{L}$ levels, the transform is done in $3\bar{L}$ iterations. Initially, a weight $w_i = 1$ is assigned to each occupied voxel i. When transforming along x, if within a block two consecutive occupied voxels are found, i.e. they store attributes $\mathbf{y}_1$ and $\mathbf{y}_2$, they are transformed according to:

$$\begin{pmatrix} DC \\ AC \end{pmatrix} = \frac{1}{\sqrt{w_1 + w_2}} \begin{pmatrix} \sqrt{w_1} & \sqrt{w_2} \\ -\sqrt{w_2} & \sqrt{w_1} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{2.10}$$

where w_1 and w_2 are their respective weights. Once done, only the weight w_1 associated to the DC component is increased by 1. Conversely, if a voxel does not need to be merged with its neighbor it is not transformed and its weight will not be updated. In this case, it can be seen as the DC coefficient is the color value stored in the occupied voxel, while the AC one its zero. The same is done along y and z directions, on all the

DC coefficients found at the previous iteration. When computed the transform along each direction, each $2 \times 2 \times 2$ block will be represented by its resulting DC coefficient, while the AC ones are ready to be entropy encoded. It is like the spatial resolution of the original SVO it's being halved in all the three dimensions, with only one voxel (the DC term) representing the original $2 \times 2 \times 2$ block. This basic transformation step is illustrated in Fig. 2.8 where, for the sake of clarity, only DC coefficients are represented.

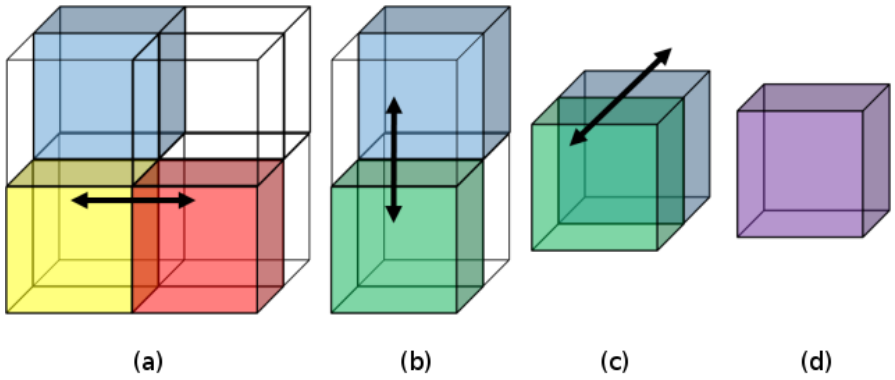


Figure 2.8.: Example of region-adaptive hierarchical transform within a $2 \times 2 \times 2$ block. [6]

In this example, red and yellow voxels are adjacent along the dimension indicated and are merged together in the green one (a). Then, green and blue have no occupied neighbors and are not transformed (b). Finally, blue and green voxels are merged (c) thus obtaining the final purple voxel (d), that will represent the $2 \times 2 \times 2$ sub-volume during when transforming the next octree level. The AC coefficient, instead will be used when decoding to restore color values between the original three occupied voxels (red, yellow and blue), starting from the purple one.
As previously said, the process is repeated on the DC terms following the spatial distribution encoded in the octree. At the end, a unique DC coefficient representing the entire volume will be obtained together with a collection of AC coefficient, needed to restore the original color attributes.
Notice that, in case w_1 and w_2 are equal (i.e. the region is regular), the transform coefficients in (2.10) become $\pm\frac{1}{\sqrt{2}}$, defining the so called *Haar Transform* already used for sub-band decomposition in images or audio. In fact, the transform performed can be interpreted as an Haar transform where different weights are assigned in order to take into account the spatial distribution of the attributes one is transforming. Moreover, each subset of transformed coefficients with the same weight can be intended as a different sub-band. For this reason, the authors called this transform *Region-Adaptive Haar Transform* (RAHT).
Once the transform is performed the coding procedure proceeds similarly to the one described in sub-section 2.2.1. Transformed coefficient are uniformly scalar quantized and entropy coded by means of an arithmetic coder, assuming a Laplacian distribution of

the coefficients. The only difference is that, in this case, each sub-band (all coefficients with the same weight) is entropy coded separately. This means that, for each sub-band they consider a Laplacian distribution:

$$p(x) = \frac{1}{2b} e^{-\frac{|x|}{b}} \tag{2.11}$$

where parameter b need to be known at the decoder to correctly reconstruct the signal. In particular, they found that the optimal value b^* that minimizes the number of bits required for encode N symbols in a sub-band is given by:

$$b^* = \frac{1}{N} \sum_{n=1}^{N} |k_n| q \tag{2.12}$$

where q is the quantization step and k_n is the value assumed by the n-th symbol within a sub-band. To encode the value b^* for each sub-band they finally used a Run Length Golomb Rice (RLGR) entropy coder.
As one can see, quantization occurs in this case as well. It is possible to evaluate the coding performance using the same Rate-Distortion curves presented before.
The author tested this procedure on $512 \times 512 \times 512$ voxel grids built upon 5 different 3D sequences with a number of occupied voxels between 200K and 300K.
They tested their algorithm against DCT and GT on all these sequences and they showed that performances of the RAHT coder are very close to the one obtained with GT (using $8 \times 8 \times 8$ block size), and sometimes are even slightly better.
It is true that increasing the block size for GT would improve compression performances (once fixed the target PSNR), but the computation times will become infeasible with real time constraints. In fact, high complexity was the main drawback of such approach. Conversely, RAHT transform requires a much lower computation to be performed. As an example, authors reported to have successfully implemented RAHT coder on common graphics processors, capable of processing frames in about 33 ms (i.e. about 30fps).

For the sake of completeness, we want to conclude this chapter mentioning that both solutions [4] and [5], were further developed, respectively in [18] and in [6], [19] to improve color coding performances when coding dynamic scenes. Both approaches focused on trying to predict color components of the actual frame starting from a reference one and using point tracking between consecutive frames and motion estimation strategies. In this way, they code only the residuals with respect to the prediction (i.e. the difference between the estimated color value for a certain voxel and its actual value), that should have a smaller dynamic than the original color signal. Also motion vectors need to be coded and sent to correctly decode predicted frames. Nevertheless, both works have showed that with inter-frame prediction they improved significantly Rate-Distortion performances. On the other side, the drawback of such approaches is the increasing of computational complexity due to tracking and motion compensation, which could be prohibitive in real time implementations. For this reason lower complexity algorithms for prediction should be considered.

3

Color coder implementation

This chapter is dedicated to the detailed description of our color coder implementation for both, static frames and dynamic sequences.

3.1 System overview

The core of our color coding system is the RAHT coder, whose behavior was already discussed in the previous chapter. We preferred this solution with respect to the others found in literature because it provides very good compression performances without requiring expensive computational effort. In particular, this characteristic was the discriminant for choosing RAHT instead of a GT approach.
The overall system is built using Matlab and C, and can be represented like in Fig. 3.1: all blocks represented by double boxes were implemented in C, the others in Matlab.

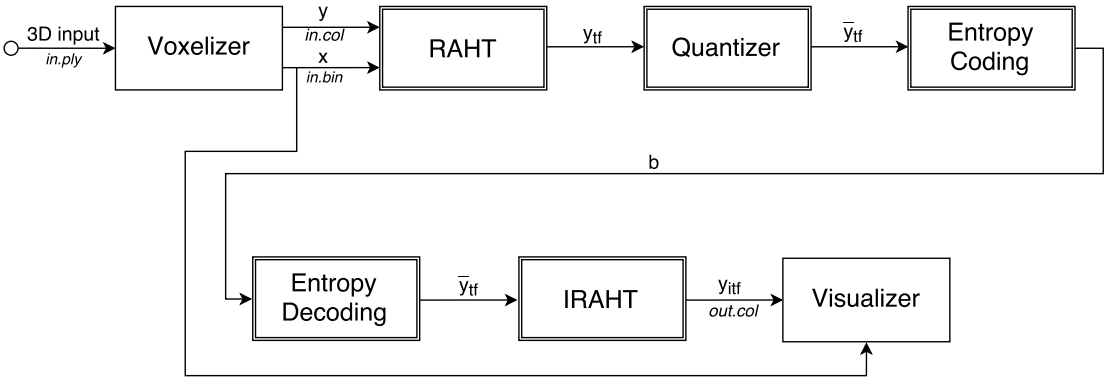


Figure 3.1.: Color coding: system overview.

Matlab blocks correspond to operations that are done offline. In a real system, these operations can be performed very quickly. This work was focused on implementing a color coder and evaluating its performances (i.e. execution time and compression).
So, let us describe how *Voxelizer* and *Visualizer* blocks work before looking at our C coder implementation. In this way the reader can understand how data are sent in input to our coder and how the output is organized.
The first block is implemented in a script called `convert_ply_bin.m`. Such a function converts pre-recorded 3D data, stored in *in.ply* w.r.t. Fig. 3.1, into their correspondent voxelized representation that consists of two files: *in.bin* and *in.col*. In this way, we can simulate a 3D acquisition system that, once acquired the point cloud, represents it as a voxel grid (see voxelization in section 2.1). The procedure must be run inside a

folder containing all *ply* files we want to convert. It scans the folder and converts each mesh found, into the correspondent *bin* and *col* files. We do not want to go very much in detail on this, nevertheless we want to highlight some aspects that should help the understanding of the procedure.

- The voxel grid size must be specified using N_{vx}, which represents the number of voxels the grid will have along each dimension (typical values are 128, 256 or 512). This operation is exactly the same as specifying the maximum decomposition level $\bar{L}$ as explained in chapter 2.

- **.bin* files are text files that store binary sequences representing the voxel grid. For this reason, each sequence is $N_{vx} \times N_{vx} \times N_{vx}$ binary symbols long and represents occupied and empty voxels as we will find them scanning the grid in raster scan order.

- **.col* files are text files that stores a triplet of 8 bit integer values in the interval $[0, 255]$ representing the RGB coordinates of each occupied voxels. RGB data are reported in the same order of the raster scanning described in the previous point. The number of bytes is equal to $3 \times n_{occ}$ (where n_{occ} is the number of occupied voxels).

- When converting a sequence of 3D models that are part of a video, we build the volume bounding box from the first frame and keep it unchanged until the end of the encoding procedure. If some object appears outside the bounding box computed upon the first frame, it will not be voxelized and coded (e.g. it's like an object outside the field of view of a camera). In this way we avoid changing the spatial resolution due to changes in the shape of the 3D object that will considerably change the dimension of the bounding box (like for example a person that opens and closes its arms).

Regarding the *Visualizer* it is implemented in the script `visualizevoxels.m`. It's a very simple script that allows 3D visualization of voxelized geometry exploiting the built-in Matlab function `scatter3.m`. This function draws colored unit circles at different 3D locations. Of course, this is only a very basic strategy: no rendering technique is applied and no other information except for geometry and color is encoded. Nevertheless, referring to Fig. 3.1, it takes in input *in.bin* (that represents geometry) and *out.col* that stores color information reconstructed at the decoder, allowing us to visualize the coding/decoding results. Moreover, it allows us to compare the visual quality degradation introduced by color quantization. In fact, optionally, one can give in input also the *in.col* file and display both models, either colored with the original or the reconstructed information.

3.2 Static frame routine

For the static encoding-decoding routine we tried two different solutions. Nevertheless, the basic steps of both implementation can be summarized as in Algorithm 1.

Algorithm 1 Coding-Decoding pipeline.

Input: *orig.bin*, dim, *orig.col*, qst, fl;

```
geom ← readGEOM(orig.bin)
color_orig ← readCOL(orig.col)
if fl then
    col ← RGBtoYUV(color_orig)
else
    col ← orig_col
end if
col ← RAHT(geom, col, dim)
quant_col ← quantize(col, fl, qst)
foo ← Arith_Code(quant_col)
quant_col2 ← Arith_Decode(foo)
col2 ← IRAHT(quant_col2)
if fl then
    col2 ← YUVtoRGB(col2)
end if
PSNR ← computePSNR(col2, color_orig)
decoded.col ← col2
```

Output: *decoded.col*, PSNR;

The algorithm takes in input *bin* and *col* files described previously, the voxel grid resolution along each axis dim, the quantization step qst, and a flag fl that if setted enables the conversion of the color space (from standard RGB to YUV) where we want to perform the transform; and returns the result of coding-decoding routine (the file *decoded.col* in Algorithm 1) together with a numerical evaluation of the distortion introduced by our system computed according to Eq. 2.9.
All inputs are passed via command line to the main function. An example of a Linux prompt command to execute the algorithm for static coding is:

```
$ ./static vase.bin 512 vase.col 10 1
```

where we suppose that our C source files have been compiled building the executable file *static*, and that we want to test our routine on a model called *vase*, voxelized in a grid of $512 \times 512 \times 512$ voxels, using a quantization step of 10 and performing the transform in the YUV color space (signaled by the last flag equal to 1).
Now, we are ready to look in detail at all the functions needed to perform such operations. We will start presenting all common aspects between the two approaches (e.g. quantization, color space transformations), and then we will illustrate the differences between the two versions with special attention to those regarding readGEOM, RAHT and IRAHT functions.

At first, we want to describe how we handle color information: colors are read through the function readCOL and are stored in a *structure*. A structure, in C programming language (but also in many other derivatives), is a user defined composite data structure that consists of a group of variables stored together under one name and that can be accessed through a single pointer. The structure is called `RGBvalue` and is made of three

double variables called R, G and B. We use double variables because, even if input data are characterized by 8 bit integer values, the transform need to process and output real valued coefficients. Moreover, we used a `struct` definition because in this way our code can be easily modified in order to handle different attributes, like texture for rendering. The process that allows loading color is illustrated in Alg. 2, where occsize is the number of occupied voxels and is assumed to be computed when reading the geometry, as we will see.

Algorithm 2 Color loading.

```
1: function READCOL(occsize, orig.col)
2:     inputcol ← FREAD(orig.col, 3occsize)
3:     for i=0 to occsize-1 do
4:         orig_col(i).R ← inputcol(i)
5:         orig_col(i).G ← inputcol(i + occsize)
6:         orig_col(i).B ← inputcol(i + 2occsize)
7:     end for
8:     return orig_col
9: end function
```

inputcol is a $3 \times$ occsize integer vector that temporary stores color attributes, read through the built-in C function FREAD from *orig.col*. Once all occsize entries in color_orig are filled it is no more needed and it is unallocated.
Then, color_orig is duplicated in a structure named col, which is processed by the following coding pipeline. The struct color_orig will be a backup of the original color components which permits computing the distortion (not necessary in a communication device).
If fl is set to 0 no transformation is applied on the color space and the three components are sent in input to the coding engine. Otherwise, the RGB color coordinates are transformed into the YUV space via the equation:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{3.1}$$

Of course, when this transformation is used, the final reconstruction of the voxel volume requires returning back to RGB coordinates at the end of the decoding procedure. This is due to two facts:

- we want to save the output of our routine in a color space suitable for visualizing the results (RGB is supported by most of the voxel visualization programs);
- we want to compute distortion in the original RGB color space.

As a matter of fact, before saving the output and computing PSNR, we apply the inverse

transformation to our decoded colors (stored in col2):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix} \tag{3.2}$$

Another operation that is implemented equally in both versions is quantization. Quantization is performed at the encoder just before converting the transformed coefficients in a bit stream. It is performed by the QUANTIZE function (w.r.t. Algo. 1) and is described in detail in Algo. 3.

Algorithm 3 Quantization.

```
1: function QUANTIZE(occsize, col, fl, qst)
2:     min = 0                                              /* 1 × 3 vector */
3:     max = 0                                              /* 1 × 3 vector */
4:     st = qst                                             /* 1 × 3 vector */
5:     if fl then
6:         st(1) = m · qst
7:         st(2) = m · qst
8:     end if
9:     for i=0 to occsize-1 do
10:        quant_col(i).R ← ROUND ( col(i).R / st(0) ) · st(0)
11:        if quant_col(i).R > max(0) then
12:            max(0) ← quant_col(i).R
13:        else if quant_col(i).R < min(0) then
14:            min(0) ← quant_col(i).R
15:        end if
16:        quant_col(i).G ← ROUND ( col(i).G / st(1) ) · st(1)
17:        if quant_col(i).G > max(1) then
18:            max(1) ← quant_col(i).G
19:        else if quant_col(i).G < min(1) then
20:            min(1) ← quant_col(i).G
21:        end if
22:        quant_col(i).B ← ROUND ( col(i).B / st(2) ) · st(2)
23:        if quant_col(i).B > max(2) then
24:            max(2) ← quant_col(i).B
25:        else if quant_col(i).B < min(2) then
26:            min(2) ← quant_col(i).B
27:        end if
28:    end for
29:    return quant_col, min, max„ st
30: end function
```

As it can be seen, the function returns an occsize long color structure with the quantized coefficients, and three 3×1 vectors containing respectively the maximum and minimum value of all three color components once quantized, and the quantization step

for each color coordinate. These vectors are used when entropy encoding the quantized coefficients, as we will see in a few lines.
The function essentially performs uniform scalar quantization according to:

$$\hat{q} = \left[\frac{q}{step}\right] \cdot step \tag{3.3}$$

If $\mathsf{fl} = 0$, the quantization step st is equal for all three RGB components and is equal to the one specified by the user via command line qst. Otherwise, if $\mathsf{fl} = 1$ and we are working in YUV color space, we use qst step size to quantize the Y component, while for U and V we increment the step st by an integer factor $textsfm \geq 0$. This is due to the fact that in YUV color space the Y component is related to luminance, while U and V refers to color information. The human eye is more sensitive to light rather than to color, so we use finer quantization for the former component and a larger quantization step for the latters. In this way we aim to improve compression without affecting the perceived quality. In the design stage we try some different values for m (i.e. $\mathsf{m} = 1, 2, 4$ and 8) and finally we choose $\mathsf{m} = 2$ considering a compression-distortion tradeoff.

Also entropy coder and decoder are implemented in the same way in both approaches, respectively in the functions Arith_Code and Arith_Decode. It should be clear that we choose an arithmetic coder for entropy compression.
For the arithmetic coding engine we used a C version based on [20]. An overview of the arithmetic coding routine is reported in Appendix A; while for a detailed implementation analysis we refer to [20]. Nevertheless, we want to describe how we used it in our routine. First of all, we want to highlight some practical aspects:

- the coder accepts in input only *positive integer values*. Unfortunately, after quantization, only integer hypothesis is verified, while coefficients are not guaranteed to be non-negative. This is why we compute min vector when performing quantization. In this way we can shift the values to be coded in order to satisfy also non-negativity requirement.

- The coder needs to know how many *different symbols* can be found in input. For this reason we compute also max vector when performing quantization. In this way one can compute the required quantity for each component as:

$$n_{symbols}(j) = \frac{\mathsf{max}(j) - \mathsf{min}(i)}{\mathsf{st}(j)} + 1, \qquad j = 0, 1, 2$$

Combining these two requirements we transform the sequence of quantized symbols according to:

$$\mathsf{quant\_col(i)}.R = \frac{\mathsf{quant\_col(i)}.R - \mathsf{min}(0)}{\mathsf{st}(0)}, \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.4}$$

$$\mathsf{quant\_col(i)}.G = \frac{\mathsf{quant\_col(i)}.G - \mathsf{min}(1)}{\mathsf{st}(1)}, \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.5}$$

$$\mathsf{quant\_col(i)}.B = \frac{\mathsf{quant\_col(i)}.B - \mathsf{min}(2)}{\mathsf{st}(2)}, \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.6}$$

in order to have $n_{symbols}(j)$ contiguous positive integer values to be coded through ARITH_CODE. According to this procedure, when decoding, st, max and min must be known and so they must be also entropy coded-decoded. Given that the three vectors are available at the decoder, one can decode the bit stream through ARITH_DECODE, obtaining the color structure that we called quant_col2. Such a structure, unless transmission errors, is equal to the encoded one, stored in quant_col. However, we want to recover the original sequence, the one resulting from quantization, not the one transformed through (3.6), (3.7) and (3.8) for entropy coding. In order to do this we must convert the entropy decoded sequence quant_col2 according to:

$$\mathsf{quant\_col2(i)}.R = \mathsf{quant\_col2(i)}.R \cdot \mathsf{st}(0) + \mathsf{min}(0), \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.7}$$

$$\mathsf{quant\_col2(i)}.G = \mathsf{quant\_col2(i)}.G \cdot \mathsf{st}(1) + \mathsf{min}(1), \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.8}$$

$$\mathsf{quant\_col2(i)}.B = \mathsf{quant\_col2(i)}.B \cdot \mathsf{st}(2) + \mathsf{min}(2), \qquad \mathsf{i} = 1, 2..., \mathsf{occsize} \tag{3.9}$$

These operations are illustrated in Fig. 3.2, where values within the circles are the ones we are interested, while the ones within the diamond box are the transformed ones, in order to meet coder/decoder requirements.

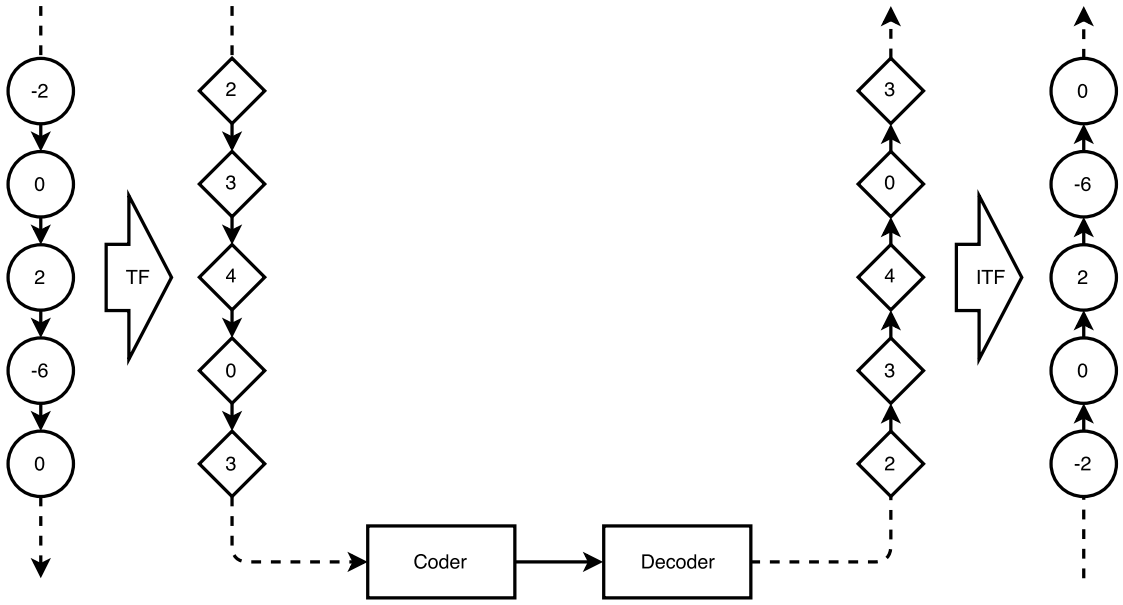


Figure 3.2.: Example of sequence transformation for entropy coding-decoding.
$\mathsf{qst} = 2$, $\mathsf{min} = -6$

In conclusion, we want to explain how we setted the arithmetic coder to encode our quantized coefficients:

- we use an *adaptive* configuration, with no assumption about the underlying coefficient distribution, i.e. at the beginning all coefficient are considered equiprobable;
- as already mentioned we use a *scalar* quantizer, i.e. we encode symbols one by one;
- we encode each color component *separately*; it means that we initialize three different instances of our arithmetic coder, and each of those will follow the distribution

of one color component.

In fact, when designing our system, we noticed that, coding color components separately (instead of using one single coder for serializing all the coefficients), increases the performances of our system in terms of both, compression performances and execution time.

Finally, the function COMPUTEPSNR computes the distortion introduced by our system using (2.9), where y_{jk} is an entry of color_orig and $\hat{y}_{jk}$ is the correspondent entry in col2. As already mentioned, we compute distortion on the RGB color space to have an idea of the distortion in the color space used to visualize our data. Moreover, before computing it, the reconstructed values stored in col2 are rounded to the nearest integer, in order to have a real estimate of the distortion. In fact, RGB values are assumed to take integer values in the interval $[0, 255]$ so, decimal values are not allowed.
Of course, other metrics are monitored in our function like execution times and the length of the generated bit stream. Nevertheless, we prefer to discuss these metrics in the following chapter, before presenting the result obtained with our coder.

3.2.1 RAHT v1

In our first implementation, we assume geometry being loaded as a dim × dim × dim integer array that stores 0 if the corresponding voxel is empty; if the voxel is occupied, the array reports its linear index, assuming the voxel grid being scanned in raster scan order as described before and illustrated in Fig. 2.2. It means that the first occupied voxel has index 1, the second is represented by index 2 and so on, until the last that will store the index occsize. This concept is illustrated in Algo. 4.

Algorithm 4 Geometry loading v1.

```
1: function READGEOM(orig.bin, dim)
2:     input ← FREAD(orig.bin, dim · dim · dim)
3:     j ← 0
4:     for z=0 to dim-1 do
5:         for y=0 to dim-1 do
6:             for x=0 to dim-1 do
7:                 if input = 1 then                    /* The voxel is occupied */
8:                     geom(x + y · dim + z · dim · dim) ← j + 1
9:                 end if
10:            end for
11:        end for
12:    end for
13:    occsize ← j
14:    return geom, occsize
15: end function
```

In this way we have a correspondence between each voxel and its associated weight during the transformation. In fact, weights are organized in an array w that stores 8 bits integer values that stores exactly occsize entries, one per occupied voxel. We do not need more than 8 bits per entry to represent weights because typical decomposition iteration values are $L = 8$ or $L = 9$ for our application; as a matter of fact, a voxel can be processed in the worst case $9 \times 3 = 27$ times (remember that in RAHT at each level we transform the color attributes 3 times, along each axis). Notice also that this function returns the value occsize, which is used by almost all functions presented before and also by the ones we are presenting in this section.
The pseudocode for RAHT is reported in Algo.5. As we can see, it returns not only the transformed coefficients col, but also the arrays mdx, msx, offs and w. These arrays include respectively the indexes of merged voxels containing the AC coefficients for a certain decomposition iteration, the indexes of voxels that store DC coefficients for the same level, the number of voxels that are transformed at a certain iteration and the final weights associated to each voxel. The former three arrays are required only by IRAHT in order to decode the transformed coefficients. In a real system this computation is not required at the encoder because the decoder can build such arrays applying RAHT to the decoded geometry, without performing color transformation (or equivalently assuming all color coordinates equal to 0). Instead, the latter one is used during the encoding, but it does not need to be returned or sent in a real implementation, because the decoder

computes it as just explained. At the end of this section, for the sake of clarity, we will report how a real decoder is able to decode color information (Algo. 9) and how it works in our implementation (Algo.8) that simulates this behavior.

Algorithm 5 RAHT v1 main function.

```
1: function RAHT(geom, col, dim, occsize)
2:     actualdim ← dim                                   /* 3 × 1 array */
3:     offs ← 0                                          /* 3 × L array */
4:     mdx ← 0                                           /* occsize array */
5:     msx ← 0                                           /* occsize array */
6:     L ← log2(dim)                                     /* Number of decomposition levels */
7:     for i=0 to 3L-1 do
8:         if i = 0 then
9:             for j=0 to occsize-1 do
10:                w(j) ← 1
11:            end for
12:            RHATITER(geom, col, w, i ÷ 3, actualdim, dim, 0, msx, mdx)
13:        else
14:            RHATITER(geom, col, w, i ÷ 3, actualdim, dim, offs(i − 1), msx, mdx)
15:        end if
16:        actualdim(i ÷ 3) ← actualdim(i÷3) / 2
17:    end for
18:    return col, mdx, msx, offs, w
19: end function
```

Let us analyze the RAHT function. All the arrays are initially set to zero, the array actualdim is equal to $(\mathsf{dim}, \mathsf{dim}, \mathsf{dim})$ and computes the number of decomposition level required L. At the first iteration w is created and all its components are initialized to 1. Then, the first transformation step is applied to the colors stored in col navigating the voxel volume geom.
The function that performs one decomposition step is called RAHTITER: it computes the transformation *in-place*, it means that it stores the transformed coefficients directly on col. We will go more in details on these aspects in a few lines when discussing about Algo. 6, 7.
Before moving, we want to notice that also a flag, computed as $\mathsf{i} \div 3$ (where $\div$ identifies the modulo operation), is passed to RAHTITER. This is required to determine along which dimension the transformation must be done:

- $\mathsf{i} \div 3 = 0$: the transform will be performed along x;
- $\mathsf{i} \div 3 = 1$: the transform will be performed along y;
- $\mathsf{i} \div 3 = 2$: the transform will be performed along z.

Once the first iteration is done, i.e. all adjacent voxels along x dimension have been merged, we update $\mathsf{actualdim}(\mathsf{i} \div 3) = \mathsf{actualdim}(0)$ halving the value it stores. This is because after this operation, we consider the voxel resolution along x axis is reduced by a factor two, because all pairs of adjacent voxels have been merged in a single DC

coefficient. After this, RAHT calls again RAHTITER, updates actualdim(i ÷ 3) and proceeds like this until the transformation is done.

Algorithm 6 RAHT v1 core function (part 1).

```
1:  function RAHTITER(geom, col, w, i ÷ 3, actualdim, dim, offs, msx, mdx)
2:      xstep ← dim/actualdim(0)
3:      ystep ← dim/actualdim(1)
4:      zstep ← dim/actualdim(2)
5:      j ← 0, x ← 0, y ← 0, z ← 0
6:      if i ÷ 3 = 0 then                                   /* Transform along x */
7:          while z ≤ dim − 1 do
8:              while y ≤ dim − 1 do
9:                  while x ≤ dim − 1 do
10:                     dx ← geom(x + xstep + y · dim + z · dim · dim)
11:                     if dx ≠ 0 then                      /* One occupied voxel */
12:                         sx ← geom(x + y · dim + z · dim · dim)
13:                         if sx ≠ 0 then                  /* It's neighbor */
14:                             dx ← dx − 1
15:                             sx ← sx − 1
16:                             a ← √(w(sx)/(w(sx) + w(dx)))
17:                             b ← √(w(dx)/(w(sx) + w(dx)))
18:                             csx ← col(sx)
19:                             cdx ← col(dx)
20:                             col(sx) ← a · csx + b · cdx
21:                             col(dx) ← a · cdx − b · csx
22:                             w(dx) ← w(dx) + 1
23:                             w(sx) ← w(sx) + 1
24:                             mdx(j+offs) ← dx
25:                             msx(j+offs) ← sx
26:                             j ← j + 1
27:                         else                            /* No neighbor found */
28:                             geom(sx) ← geom(dx)
29:                         end if
30:                     end if
31:                     x ← x + 2 · xstep
32:                 end while
33:                 y ← y + ystep
34:             end while
35:             z ← z + zstep
36:         end while
```

Algorithm 7 RAHT v1 core function (part 2).

```
37:     else if i ÷ 3 = 1 then                                   /* Transform along y */
38:         while z ≤ dim − 1 do
39:             while x ≤ dim − 1 do
40:                 while y ≤ dim − 1 do
41:                     dx ← geom(x + (y + ystep) · dim + z · dim · dim)
42:                     if dx ≠ 0 then                           /* One occupied voxel */
43:                         sx ← geom(x + y · dim + z · dim · dim)
44:                         if sx ≠ 0 then                       /* It's neighbor */
45:                             ...
46:                         else
47:                             ...
48:                         end if
49:                     end if
50:                     y ← y + 2 · ystep
51:                 end while
52:                 x ← x + xstep
53:             end while
54:             z ← z + zstep
55:         end while
56:     else                                                     /* Transform along z */
57:         while y ≤ dim − 1 do
58:             while x ≤ dim − 1 do
59:                 while z ≤ dim − 1 do
60:                     dx ← geom(x + y · dim + (z + zstep) · dim · dim)
61:                     if dx ≠ 0 then                           /* One occupied voxel */
62:                         sx ← geom(x + y · dim + z · dim · dim)
63:                         if sx ≠ 0 then                       /* It's neighbor */
64:                             ...
65:                         else
66:                             ...
67:                         end if
68:                     end if
69:                     z ← z + 2 · zstep
70:                 end while
71:                 x ← x + xstep
72:             end while
73:             y ← y + ystep
74:         end while
75:     end if
76:     offs(i) ← offs(i − 1) + j
77: end function
```

As already explained, the flag $i \div 3$ identifies along which dimension the transform must be performed (checking the statements at lines 6, 37, and 57). Once identified, it

navigates the voxel grid stored in geom and performs the transform described in section 2.2.2. Essentially, the transform operations are the same in all three cases (e.g. computation of A,B, weights updates etc...). For this reason we report their implementation only in Algo. 6 and we avoid repeating the same code in 7, where we replaced it with dots.
In order to help understanding how RAHTiter works, we want to make a few observations about the pseudocode reported:

- in order to correctly scan the voxel grid represented by geom we need to compute xstep, ystep and zstep at each call. This is because, after each iteration, the resolution of the axis along which the transform was performed is halved. In fact, we process pairs of voxels at a time but at the next iteration only DC coefficient of each couple must be considered by the transform. So, for example at the first call we have that xstep = ystep = zstep = 1, while at the second one we have ystep = zstep = 1 and xstep = 2. In fact, when transforming for the first time along y we have already computed one iteration along x: it means x resolution is halved and so we must scan geom with a doubled step size along that dimension (i.e. at first iteration we check each voxel while at the second one we check only one every two). This process is repeated along each dimension (once processed we double the correspondent step) until the voxel grid has resolution $1 \times 1 \times 1$ and xstep = ystep = zstep = 512.

- The variable that corresponds to the dimension in which the transform is being performed is incremented with a double step, while the others, one step at a time. This is because along the transform dimension we check couples of voxels so, once again considering the first decomposition along x, if we have checked voxel 0 and 1, we must skip to voxels 2 and 3 (i.e. we must increment x by $2 \cdot$ xstep).

- When performing the transform between two adjacent occupied voxels, we store the DC coefficient in col index pointed by sx the AC one to the one pointed by dx.

- When performing the check on dx and sx we consider linear indexes in the interval $[1, \text{occsize}]$ in order to distinguish occupied voxels as the ones different from zero. If both voxels are occupied (i.e. the transform must be computed), such indexes are decremented by one unit in order to be aligned to array indexing in C that starts from 0. In this way we can recover the correct weights w and save results correctly inside col.

- Adjacent voxel at a certain iteration are far away xstep positions inside geom when transforming along x, ystep $\cdot$ dim along y and zstep $\cdot$ dim $\cdot$ dim along z.

- When checking a pair, we look first at the next voxel (dx) rather than to the actual one (sx) because in this way we save a lot of useless comparisons. In fact, if dx is empty we can skip directly to the next pair, because sx could be either empty or occupied, but in both cases it doesn't need to be processed. Doing the opposite, we should have checked both sx and dx. Of course, its related with our choice of storing DC coefficients in sx and AC ones in dx. If, instead dx is occupied we check sx: if it is occupied the couple is transformed, otherwise the index stored in dx is copied to sx to be transformed during successive iterations.

Assuming mdx, msx, offs, w and dim being available at the decoder, the transform can be inverted by IRAHT as reported in Algo. 8.

Algorithm 8 IRAHT.

```
1: function IRAHT(quant_col2, dim, w, msx, mdx, offs)
2:     L ← log2(dim)                                   /* Number of idecomposition levels */
3:     for i= 3L-1 to 0 do
4:         for j = offs(i) to MAX(offs(i − 1), 0) do
5:             sx ← msx(j)
6:             dx ← mdx(j)
7:             w(sx) ← w(sx) − 1
8:             w(dx) ← w(dx) − 1
9:             a ← sqrt(w(sx)/(w(sx) + w(dx)))
10:            b ← sqrt(w(dx)/(w(sx) + w(dx)))
11:            csx ← quant_col2(sx)
12:            cdx ← quant_col2(dx)
13:            quant_col2(sx) ← a · csx − b · cdx
14:            quant_col2(dx) ← a · cdx + b · csx
15:        end for
16:    end for
17:    col2 ← quant_col2
18:    return col2
19: end function
```

Basically, the functions applies inverse transform to each couple of voxels that was transformed (into AC and DC components) at each RAHTiter call. They are stored in msx(j) and msx(j), while offs is required to correctly navigate inside those two arrays. In fact, for example all DC coefficients computed at the $i-th$ iteration are stored between msx(offs(i)) and msx(offs(i − 1)). The only exception regards the coefficients of the first iteration that are stored between msx(offs(i)) and msx(0). This is why we used the MAX function in line 4.
As just said, in our routine those quantities are computed at the encoder and then passed to the decoder. However, this information does not need to be encoded and transmitted, because can be recovered at the decoder, once the geometry is reconstructed as illustrated in Algo. 9. Only dim needs to be encoded and sent.

Algorithm 9 IRAHTreal.

```
1: function IRAHTreal(geom, quant_col2, dim, occsize)
2:     zero_col ← 0                                    /* 1 × occsize RGBvalue struct */
3:     w, msx, mdx, offs ← RAHT(geom, zero_col, dim, occsize)
4:     col2 ← IRAHT(quant_col2, dim, w, msx, mdx, offs)
5:     return col2
6: end function
```

Regarding Algo. 9, it is worth highlighting that the structure zero_col is used in input to RHAT because, in this case, we don't want to perform the transform, but only to

recover arrays required for decoding.

3.2.2 RAHT v2

As we will see in the next chapter, the system described in subsection 3.2.1 works quite well for low resolution voxel grids (until $\mathsf{L} = 7$ or 8). Instead, when increasing the number of voxels, it slows down significantly because of the high number of checks performed during the first iterations, where we visit the voxel cube at full resolution.
For this reason we decided to develop a different procedure to navigate the grid, trying to avoid scanning large empty sub-volumes. In order to do this we decided to load geometry coordinates differently from before: we organize occupied voxels in a C structure that we called `tosort`, composed by two integer values:

- *idx*: stores the linear coordinate of each occupied voxel according to some grid scanning order;
- *colidx*: stores occupied different integer values in the interval $[1, \mathsf{occupied}]$.

In Algo. 10 we report how this structure is filled, i.e. how we load and represent geometry in this version of the transform.

Algorithm 10 Geometry loading v2.

```
1:  function readGEOM(orig.bin, dim)
2:      input ← FREAD(orig.bin, dim · dim · dim)
3:      j ← 0
4:      for i=0 to dim · dim · dim − 1 do
5:          if input = 1 then                          /* The voxel is occupied */
6:              geom(j).idx ← i + 1
7:              geom(j).colidx ← j
8:          end if
9:      end for
10:     occsize ← j
11:     return geom, occsize
12: end function
```

In this way, we store linear coordinates of all the occupied voxels, that can take values in the interval $[1, \mathsf{dim} \cdot \mathsf{dim} \cdot \mathsf{dim}]$, in an $1 \times \mathsf{occsize}$ `tosort` structure.
Instead, in the other variable inside the structure, we save indexes of such voxels similarly as done in the previous method: the first occupied voxel during the loading operation will have the correspondent *colidx* field set to 0, the second will have $colidx = 1$ and so on, until the last that will have $colidx = \mathsf{occsize} - 1$. This representation, even if more complex than the previous one (it requires a structure in place of an array), is much more compact because it contains only occupied voxels, whose number is typically much lower than the total number of voxels in the whole grid.
In order to understand how we navigate this structure to perform the transform, let us review the coding process for RAHT v2 (Algorithm 11).

Algorithm 11 RAHT v2 main function.

```
1: function RAHT(geom, col, dim, occsize)
2:     actualdim ← dim                                   /* 3 × 1 array */
3:     offs ← 0                                          /* 3 × L array */
4:     mdx ← 0                                           /* occsize array */
5:     msx ← 0                                           /* occsize array */
6:     L ← log2(dim)                                     /* Number of decomposition levels */
7:     for i=0 to 3L-1 do
8:         if i = 0 then
9:             for j=0 to occsize-1 do
10:                actualdim(i ÷ 3) ← actualdim(i÷3) / 2
11:                w(j) ← 1
12:            end for
13:            RHATITER(geom, col, w, i, actualdim, occsize, msx, mdx, 0)
14:            QSORT(geom, occsize)
15:        else
16:            RHATITER(geom, col, w, i, actualdim, occsize, msx, mdx, offs(i − 1))
17:            QSORT(geom, occsize − offs(i-1))
18:        end if
19:    end for
20:    return col, mdx, msx, offs, w
21: end function
```

It is similar to the previous one, so we will focus only on the two main differences:

1) RAHTITER requires occsize instead of dim as input parameter, while the others are the same (even if geometry is represented differently and the flag is i instead of i ÷ 3);

2) after each RAHTITER call we perform a sorting operation (on *idx*). The reason why it is done will be clear when looking at how geometry is scanned (Algo. 12 and 13). Nevertheless, the sorting operation is performed using the QSORT built in C function. At each iteration the routine sorts the last occsize − offs(i-1) entries of geom in increasing order. The only exception is the first one where we need to sort the entire structure.

Looking at the new version of RAHTITER 12, one can see that there are a lot of operations that are performed exactly as before: the transform, of course, is the same, AC and DC coefficients are stored in a similar way (e.g. AC is stored in the col index identified by dx) and also mdx msx, w and offs are similarly computed.
However, one can also see that geometry scanning is very different. For example, there is no need to compute xstep, ystep and zstep, there are no nested `While` loops, there are no three different cases depending on which dimension we are transforming. Moreover, the check performed in order to find pair of voxels is different and, finally, a function is introduced in this version:UPDATECOOR.

Algorithm 12 RAHT v2 core function

```
1: function RAHTITER(geom, col, w, i, actualdim, occsize, msx, mdx, offs)
2:     j ← 0, k ← 0,
3:     while k + offs ≤ occsize − 1 do
4:         sxc ← goem(k + offs).idx
5:         dxc ← goem(k + offs + 1).idx
6:         if (sxc÷ = 1) & (sxc = dxc − 1) then        /* Two adjacent voxel found */
7:             sx ← geom(offs+k).colidx
8:             dx ← geom(offs+k+1).colidx
9:             a ← √(w(sx)/(w(sx) + w(dx)))
10:            b ← √(w(dx)/(w(sx) + w(dx)))
11:            csx ← col(sx)
12:            cdx ← col(dx)
13:            col(sx) ← a · csx + b · cdx
14:            col(dx) ← a · cdx − b · csx
15:            w(dx) ← w(dx) + 1
16:            w(sx) ← w(sx) + 1
17:            mdx(k+offs) ← dx
18:            msx(k+offs) ← sx
19:            goem(k + offs + 1).idx ← 0
20:            goem(k + offs).idx ← UPDATECOOR(goem(k + offs).idx, i, actualdim)
21:            j ← j + 1
22:            k ← k + 1
23:        else                                        /* No neighbor found */
24:            goem(k + offs).idx ← UPDATECOOR(goem(k + offs).idx, i, actualdim)
25:        end if
26:        k ← k + 1
27:    end while
28:    offs(i) ← offs(i − 1) + j
29: end function
```

As done in the previous section, we want to go in detail in some aspects of RAHTITER in order to help the reader understanding the pseudocode reported:

- at each iteration, in order to scan geometry, the function scans exactly occsize − offs(i − 1) entries one by one starting from offs(i − 1) (or from 0 at the very first call). In the pseudocode we omitted reporting the index of the array offs in order to avoid being too heavy in our notation. Each time we referred to offs we intended offs(i − 1) except in line 28 where we have explicitly expressed the indexes to avoid confusion.

- In order to find adjacent voxels, the routine looks at *idx* values of the actual and subsequent geom entries. If the index stored in the actual entry corresponds to an odd coordinates l (i.e. it is the first one of a pair of occupied voxels, if such a pair exists) the routine checks *idx* field of the next one. If it stores the index $l + 1$ the voxels are adjacent and are merged. Otherwise, it updates the actual linear coordinate through UPDATECOOR as we will see, and restarts the procedure from

the next entry. Notice that, in this case all voxels stored in geom are occupied, so our check consists in verifying whether two subsequent entries corresponds to adjacent voxel at a certain iteration.

- If two voxels need to be merged, we need to look at their *colidx* to find the correct col and w entries to perform the transform. In fact, while at the first iteration geometry and color structures are aligned (i.e. the first entry in col corresponds the first one in geom and so for all the others), in the other ones this is no more true. Geometry coordinates are updated (when performing the transform) and sorted (after each call of RAHTiter) in increasing order according to *idx* in order to scan the volume and look for adjacent pairs along another dimension. So we need the field *colidx* in order to keep trace of the voxels' location at the beginning of the procedure (i.e. correctly access col and w).

- As already anticipated, the goal of updateCOOR is to express the actual voxel linear coordinate (let say l_x for example assuming scanning the volume along x) in the linear coordinate l_y it will have in the voxel grid at the next iteration. For this reason, it must be applied only to sx voxels (the one that stores DC coefficients to be further processed). Moreover, it must take into account that at each iteration the actual scanning dimension is halved. The process that allows to correctly updates coordinates is reported in Algo. 13.

- *idx* field that corresponds to dx voxels are set to zero once the transform is done. This is because at the next iteration we don't need to check that positions that stores AC blocks. In fact, after sorting all these entries will occupy the first $\mathsf{offs}(\mathsf{i}-1)-1$ entries of geom (remember that we will restart our scansion at $\mathsf{offs}(\mathsf{i}-1)$).

- k is incremented at each iteration (in order to visit consecutive entries) but, when transform is performed, its value is increased also at line 22 of Algorithm12. In this way the routine jumps two entries ahead, i.e. it skips checking the next voxel because it has already been processed in the actual iteration.

Algorithm 13 Updating linear coordinates.

1: **function** changeCOOR(x.idx, actualdim, i)
2: $\quad \mathsf{x}.idx \leftarrow \lceil \mathsf{x}.idx/2 \rceil$
3: $\quad \mathsf{tmp} \leftarrow \mathsf{actualdim}((\mathsf{i}) \div 3) \cdot \mathsf{actualdim}((\mathsf{i}+1) \div 3)$
4: $\quad \mathsf{div} \leftarrow \lfloor (\mathsf{x}.idx - 1)/\mathsf{tmp} \rfloor$
5: $\quad \mathsf{dim1} \leftarrow \mathsf{div} \cdot \mathsf{actualdim}((\mathsf{i}+1) \div 3)$
6: $\quad \mathsf{mod} \leftarrow (\mathsf{x}.idx - 1) \div \mathsf{tmp}$
7: $\quad \mathsf{dim2} \leftarrow \lfloor \mathsf{mod}/\mathsf{actualdim}(\mathsf{i} \div 3) \rfloor$
8: $\quad \mathsf{dim3} \leftarrow \mathsf{mod} \div \mathsf{actualdim}(\mathsf{i} \div 3) \cdot \mathsf{actualdim}((\mathsf{i}+1) \div 3) \cdot \mathsf{actualdim}((\mathsf{i}+2) \div 3)$
9: $\quad \mathsf{x}.idx \leftarrow \mathsf{dim1} + \mathsf{dim2} + \mathsf{dim3} + 1$
10: $\quad$ **return** x
11: **end function**

Finally, we need to explain how *idx* values are updated. First of all, notice that actdim(i ÷ 3) value is updated at the beginning of the `For` loop in Algo. 11. This is

because, during a transform iteration, UPDATECOOR updates coordinates for the next one, so it need to know which will be the dimensions of the voxel grid once the transformation along the actual dimension is completed.
At the first step, the function computes the coordinates that a point x will have at the next iteration considering only the fact that the grid resolution will be halved (in the actual transforming dimension). We apply $\lceil \cdot \rceil$ operation because we start enumerating linear coordinates starting from one; for example, voxel with coordinates 1 and 2, along the actual dimension, will have both coordinate 1 within the new voxel volume, assuming it will be scanned in the same order.
Other operations, instead, are needed in order to express the updated coordinates, assuming a new scanning order, which allows finding adjacent voxels along another dimension. To make it clear, we want to make an example using the simple $4 \times 4 \times 4$ voxel grid presented in chapter 2, reported for convenience in Fig. 3.3.

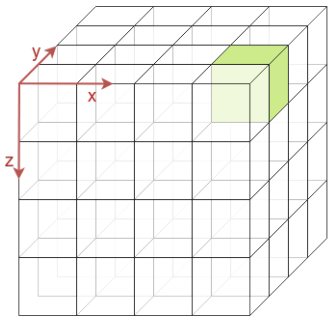


Figure 3.3.: $4 \times 4 \times 4$ voxel grid.

In this example, the only occupied voxel has linear coordinate $l_x = 8$, assuming the cube being traversed by moving at first along x, then along y and, finally, along z. We also assume the first voxel having coordinate 1. Instead, if we scan the cube moving along y, z and x, the linear coordinate is $l_y = 50$. In case we scan it in z, x and y order we have $l_z = 29$. We report these three configurations, because they are the ones implemented in our function. When expressing linear coordinates using the order we used for l_i we are looking for adjacent voxels along i dimension ($i = x, y, z$). We use these particular configurations as they represent a rotation of the scanning order along the three axis. The same procedure can be used to update coordinates at each iteration. In fact, denoting the scanning order as xyz for l_x, yzx for l_y and zxy for l_z, we can see that, basically, we are circularly shifting the axis order. Moreover, starting from z, we obtain again the configuration xyz that is, in fact, the correct order to proceed with the transform.
Given this, we want to illustrate how UPDATECOOR converts linear coordinates from a reference scanning order to the next one. For the sake of clarity, we will illustrate this process when changing from xyz to yzx order.
In lines 3 and 4 it computes how many xy voxel planes we have before finding the one that contains our considered voxel (in our example $div = 0$). In other words the function computes z coordinate, which must be multiplied by the resolution along y dimension. In fact, in the next scan, when we increase the value of z, it means that we have completed the scansion of a row of voxels along y. Note that in line 4 (but also in line 6) x.idx is decremented of a unit. This is because the change works if our linear coordinates starts

from 0, not from 1 as in our case. For this reason, we shift down the values of a unit, compute the transform and re-add a unit when computing the final new coordinate in line 9.
Line 7 computes the y displacement that we do not need to multiply because at the next iteration we will navigate along that direction. Finally, line 8 computes the number of yz planes we must scan before finding the one containing the voxel of interest and multiplies that quantity by the resolution along y times the resolution along z. In our example following this procedure we have $\mathsf{dim1} = 0$, $\mathsf{dim2} = 1$ and $\mathsf{dim1} = 48$ and so $l_y = 50$ as seen before.
Notice that, idx values, once updated in this way, need to be sorted, because we need to scan them in increasing order to find adjacent ones as described before. We want to conclude saying that, regarding IRAHT and IRAHTreal functions, the routines and the consideration are exactly the same of the previous section (obviously in this case we apply our new version of RAHT when decoding), so we refer to Algo.9, 8 and relative comments.

3.3 Dynamic sequence routine

In order to exploit temporal redundancy between consecutive frames in a 3D video sequence, we developed a prediction strategy that aims to improve compression without being too expensive, from a computational point of view. Before illustrating our prediction strategy, we want to show how the overall system processes sequences of 3D frames. The system overview is reported in Algo. 14.

Algorithm 14 Prediction routine overview.

Input: dim, qst, fl, nframes, r, gof;

```
1:  nbits ← 0
2:  PSNR ← 0
3:  for j from 0 to nframes-1 do
4:      statfl ← 1
5:      geom ← readGEOM(frame_j.bin)
6:      color_orig ← readCOL(frame_j.col)
7:      if fl then
8:          col ← RGBtoYUV(color_orig)
9:      else
10:         col ← orig_col
11:     end if
12:     if (j ÷ gof ≠ 0) & j ≠ 0 then
13:         statfl ← 0
14:     end if
15:     if statfl then                          /* Code frame without prediction */
16:         deco, len ← RAHTcoder(geom, col)
17:     else                                    /* Code frame using prediction */
18:         diff ← predict(3Dcol, col, geom, r)
19:         TFdiff, len ← RAHTcoder(geom, col)
20:         deco ← predictINV(3Dcol, TFdiff, geom, r)
21:     end if
22:     3Dcol ← 3Dstore(deco)
23:     if fl then
24:         deco ← YUVtoRGB(deco)
25:     end if
26:     DIST ← DIST + computePSNR(deco, color_orig)
27:     nbits ← nbits + len
28:     decoded.col ← append(decoded.col, deco)
29: end for
```

Output: nbits/nframes, DIST/nframes, *decoded.col* ;

The block that performs transform and its inverse on each frame is the function RAHTcoder. Basically, it implements the static frame coding routine described in the previous section (Algo. 1 with RAHT v2) made exception for some detail that we are going to highlight:

- it takes as input geom and col already loaded as structs, while other inputs are the same listed in Algo. 1 (we have not listed all of them in lines 16 and 19);
- it does not perform PSNR computation, color space transformation and output saving, because these operations are performed in the main routine too;
- it returns, instead, the bit size of the bit stream generated for each frame (len), and the color structure containing decoded coefficients (deco).

Quantization, entropy coding and all the other operations described before are exactly the same.
Algorithm 14 takes in input dim, qst, fl that are needed by RAHTcoder to perform encoding-decoding routine (of course together with geom and col). In addition, other tree parameters need to be specified when running our prediction coding scheme:

- nframes, which is the number of frames in the sequence we want to process;
- r, which is a parameter of our prediction scheme, as we will see;
- gop, which allows us to set the number of frames in a so called *Group Of Pictures*.

Our basic GOP structure consists in: an I frame, which is the first one and can be decoded by itself (i.e. no prediction was used while encoding it); $\mathsf{gop} - 1$ *P frames*, which can be decoded only starting from the correspondent reference one (i.e. they were encoded using prediction starting from I). GOPs structures are widely used in all actual 2D video codecs and are required, when a temporal prediction scheme is applied, in order to enable pseudo-random access to frames when decoding and mitigate the error propagation in case of losses. In fact, for example, by inserting a reference frame every gop ones, the user can access the decoded sequence randomly, i.e. decoding the sequence from the nearest previous I frame, not from the beginning of the whole sequence.
In this routine, no *.bin* and *.col* files are specified as inputs. In fact, our code is intended to be executed inside a folder where successive voxelized 3D frames are stored as $frameXXXX.bin$ and $frameXXXX.col$ where XXXX are the number of the frame in the considered sequence, starting from 0000 (in the pseudocode we indicate the frame read at a certain iteration with $frame\_j$). At this point, we want to report an example of Linux prompt command to run an instance of our code, assuming our source code being compiled building the executable *prediction*:

```
$ ./prediction 512 10 1 150 2 30
```

The first four arguments (including the executable file), have already been discussed when analyzing static routine, while the others means that we want to encode the first 150 frames of our input sequence inserting a reference frame every 30 and predicting with parameter $\mathsf{r} = 2$.
Before looking at how predict and predictINV works and explain the meaning of r, we want to briefly sum up which operations are done in the main function.
Prediction loops among all frames we want to process: at each loop the flag statfl is initially set to 1 (i.e. the frame will be coded without prediction). Geometry and color attributes are read and, if needed, color space conversion is applied. Line 12 checks if the actual frame must be encoded using prediction (i.e. if it is not the first one or a

multiple of gof). If the check succeeds statfl is resetted to 0. Depending on the value of statfl the frame is either directly processed through RAHTcoder or predicted from the previous one. In the latter case, the prediction residual for each voxel is computed (i.e. the difference between the actual color values and the predicted ones) and it will be sent as input to the RAHTcoder. Then, the decoded residuals TFdiff are added to the prediction we made from the previous decoded frame in order to restore color attributes: such operation is implemented in predictINV. In both cases (when using prediction or not), the procedure saves a copy of the actual frame color through 3Dstore in a dim × dim × dim RGBvalue struct that stores attributes keeping track of their spatial distribution. In fact, 3Dcol represents a voxel grid, scanned in xyz order with the same orientation of Fig. 3.3, where each $(0, 0, 0)$ entry corresponds to an empty voxel, while the others represents RGB or YUV values of that specific occupied voxel. In this way, at the next iteration, the routine can easily make temporal prediction, if needed. Conversely, using linear coordinates, it will be more difficult using our prediction strategy.
Then, decoded colors are re-converted in RGB values, if needed, in order to compute PSNR. The sum of all PSNR values is finally stored in the variable DIST, in order to return the average PSNR value of the whole sequence. Similarly, nbits stores the sum of all the number of bits required for encoding the sequence, in order to compute the average bit rate per frame.
Finally, the file *decoded.col* contains all decoded color values. It is generated by appending the decoded colors for the actual frame to the ones computed for the previous ones and already stored in *decoded.col*.
About our prediction strategy, we implemented a simple prediction scheme, exploiting the fact that between consecutive frames we can assume limited motion. It means that an occupied voxel in the actual frame very probably corresponds to the same occupied voxel in the previous one (i.e. has the same 3D coordinates) or to a previous voxel within a small 3D neighborhood of the actual one. In this way we can estimate the color attribute of the actual voxel looking at the one its correspondent has in the previous frame. In fact, we can assume that color attributes of correspondent occupied voxels (i.e. representing the same part of the voxelized object) do not change so much from one frame to the other. For these reasons, we developed a prediction routine that can be summarized as follows:

- for each occupied voxel that has a matching voxel in the previous frame, code the difference between actual color value and the previous one (they should have very close values);
- for each occupied voxel that doesn't have a correspondent in the previous frame, search in an $(2\mathsf{r}+1) \times (2\mathsf{r}+1) \times (2\mathsf{r}+1)$ neighborhood within the previous frame if there are occupied voxels;
- if neighborhood is not empty, code the difference between the actual color value and the mean of all neighbors found;
- if no neighbor is found, skip the prediction and encode the color attribute as *intra*.

This last case can happen when dealing with noisy acquisitions, fast moving objects or when a new object appears in our 3D voxel grid. This process done by the function predict whose behavior is described in Algorithm 15.

Algorithm 15 Prediction.

```
1: function PREDICT(3Dcol, col, geom, r, occsize, dim)
2:     for i = 0 to occsize − 1 do
3:         cnt ← 0
4:         sum ← 0                                        /* 1 × 1 RGBvalue struct */
5:         if 3Dcol(geom(i.idx) − 1) = 0 then
6:             for x = −r to r do
7:                 for y = −r to r do
8:                     for z = −r to r do
9:                         aux ← geom(i.idx) − 1 − x − y · dim − z · dim · dim
10:                        if 0 ≤ aux < dim · dim · dim then
11:                            sum ← sum + 3Dcol(aux)
12:                            cnt ← cnt + 1
13:                        end if
14:                    end for
15:                end for
16:            end for
17:            if cnt = 0 then                            /* No neighbor found */
18:                diff ← col(i)
19:            else
20:                diff ← col(i) − sum/cnt
21:            end if
22:        else
23:            diff ← col(i) − 3Dcol(geom(i.idx) − 1)
24:        end if
25:    end for
26:    return diff
27: end function
```

It takes in input the actual geometry distribution geom and color attributes col, together with the actual number of occupied voxels (occupied), the voxel grid resolution dim and the previous frame color attributes stored as explained before in 3Dcol in order to easily look for matching voxels.
The function loops among all the actual occupied voxels and for each of these it checks whether an occupied voxel with the same coordinates is present in 3Dcol or not (line 5). If present, the function skips to line 23 where we compute the residual between actual and previous color information. Notice that, even if not reported explicitly in the pseudocode, all operations between `RGBvalue` structures are intended to be performed for all their three components (R, G and B) independently.
If not present, the function starts looking for occupied voxels in 3Dcol scanning a $(2\mathsf{r}+1) \times (2\mathsf{r}+1) \times (2\mathsf{r}+1)$ neighborhood centered on the actual coordinate. An example of such a neighborhood is reported in in Fig. 3.4 where $\mathsf{r} = 1$.

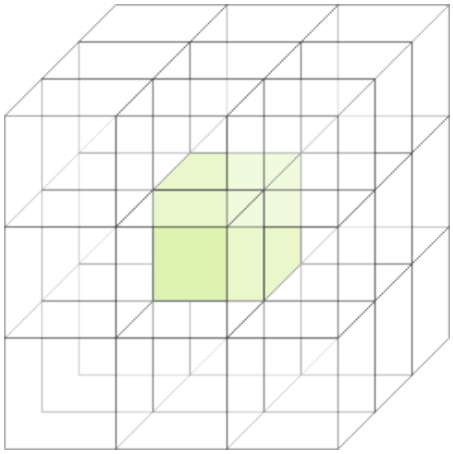


Figure 3.4.: Prediction patch: $\mathsf{r} = 1$.

Within the prediction patch, it sums all color values of the occupied voxels found in sum and counts them in cnt.
Here we want to mention that, the check in line 10 is required in order to avoid overflow problems. In fact, if the actual voxel is located near the boundary of our voxel grid (i.e. one of its 3D coordinate is greater than $\mathsf{dim} - \mathsf{r}$ or is less than r, assuming coordinates expressed in the interval $[1, \mathsf{dim}]$) the neighbor search will go outside the grid (i.e. aux will point to an index that is not valid for accessing 3Dcol).
Nevertheless, if no occupied voxel is found ($\mathsf{cnt} = 0$), we skip prediction. Otherwise, our prediction is the mean of all color attributes within the $(2\mathsf{r}+1) \times (2\mathsf{r}+1) \times (2\mathsf{r}+1)$ region, and once again it computes the prediction residuals.
The function returns diff, which stores prediction residuals or actual color values (if prediction is skipped) and will be transformed through RAHTcoder.
This procedure must be inverted at the decoder in order to restore color attributed from decoded residuals TFdiff. This operation is performed by the function predictINV that computes prediction for each voxel as in predict and adds to it the corresponding decoded residual stored in TFdiff. The pseudocode that illustrates this process is reported in Algo. 16. We do not want to analyze it in detail because essentially it works like our prediction routine.
It is possible to conclude that this color prediction scheme can be seen as an alternative to the ones already present in literature (based on voxel tracking, motion estimation and compensation), since it provides similar compression performances without being so computational demanding. Moreover, this method is not affected by the fact that number of occupied voxels changes at every frame, that is one of the major issues one must face when tracking voxels in consecutive frames.
The results obtained with this prediction method will be presented in chapter 4.

Algorithm 16 Prediction decoding.

```
1: function predictINV(3Dcol, TFdiff, geom, r, occsize, dim)
2:     for i = 0 to occsize − 1 do
3:         cnt ← 0
4:         sum ← 0                                          /* 1 × 1 RGBvalue struct */
5:         if 3Dcol(geom(i.idx) − 1) = 0 then
6:             for x = −r to r do
7:                 for y = −r to r do
8:                     for z = −r to r do
9:                         aux ← geom(i.idx) − 1 − x − y · dim − z · dim · dim
10:                        if 0 ≤ aux < dim · dim · dim then
11:                            sum ← sum + 3Dcol(aux)
12:                            cnt ← cnt + 1
13:                        end if
14:                    end for
15:                end for
16:            end for
17:            if cnt = 0 then                              /* No neighbor found */
18:                deco ← TFdiff(i)
19:            else
20:                deco ← TFdiff(i) + sum/cnt
21:            end if
22:        else
23:            deco ← TFdiff(i) + 3Dcol(geom(i.idx) − 1)
24:        end if
25:    end for
26:    return deco
27: end function
```

4

Experimental results

In this chapter we want to describe our experimental setup and to show the results obtained with our coder.

Our experiments were run on a workstation with the following characteristics:

OS version	Microsoft Windows 10 Pro
Memory	32GB DDR4-2400 @1200 MHz
Processor	Intel Core i7 7700 CPU @3.60GHz
Storage	256GB SSD, 2TB 7200RPM
Graphics	Intel HD630 NVIDIA GeForce GTX 1070

Table 4.1.: Workstation's specifications.

The executables were built using `MinGW` that is a free and open source development environment for creating MS Windows applications. Basically, it consists in a porting of the famous `Gnu Compiler Collection` (GCC) Linux compiler. The version used is the last one available, that corresponds to `gcc-6.3.0`. We compiled our source files using the *-Ofast* option. Essentially, it enables almost all the compiler optimizations, in order to improve run-time performances (in particular we are interested on reducing the execution time). Given this, a typical prompt command line to compile our sources looks like:

```
$ gcc RAHT.c ac.c ac.h -Ofast -o coder
```

where *RAHT.c*, *ac.h* and *ac.c* are respectively our C source code (the one that implements RAHT), the arithmetic coder interface and implementation, while *coder* is the executable file generated.
For measuring the execution times we use function CLOCK() of the standard C library `<time.h>`. In particular, we measured:

- **PT**: the time required to compute prediction (in case of video sequences);
- **TT**: the time required to compute RAHT transform, together with quantization;
- **ET**: the time required to entropy encode data.

Instead, we don't considered times required to load geometry and color data for each frame nor for color space conversion (when performed).
About compression result, the only aspect we want to highlight is the fact that, in the

computation of the rate required to encode data, it is not considered the overhead due to quantization parameters that needs to be transmitted to the decoder. Nevertheless, especially when encoding video sequences, that amount of bits can be neglected.
Finally, as just anticipated, we compute distortion between original RGB attributes and reconstructed ones through (2.9), rounding the reconstructed ones to the nearest integer in the interval $[0, 255]$: no rendering technique is applied in our evaluations.

4.1 Static models

At first we want to show the dataset used to test static model encoding. We used three different 3D models: *Girl*, *Vase* and *HockeyPlayer*. These voxelized models are represented respectively in Fig. 4.1, Fig. 4.2 and in Fig. 4.3. In Fig. 4.2b, we reported also a detailed view of the same model, in order to appreciate the result of the voxelization procedure.

(a) Front.

(b) Back.

Figure 4.1.: Girl: $L = 9$

(a) Extended view.

(b) Voxelization detail.

Figure 4.2.: Vase: $L = 9$

(a) Front.

(b) Back.

Figure 4.3.: HockeyPlayer: $L = 9$

We choose these three models, among the one we have, because of their different number of occupied voxels (n_{occ}). In fact, as can be seen in Table 4.2, *Vase* has a number of occupied voxels greater than *HockeyPlayer* that, in turn, has n_{occ} significantly greater than *Girl*, for all considered values of L. This allows us to compare our static frame routines, not only for different L values, but also for different n_{occ}, given L.

Model	**Octree depth**	n_{occ}	n_{occ} (%)
Girl	$L = 7$	9643	0.5
	$L = 8$	37489	0.2
	$L = 9$	140236	0.1
HockeyPlayer	$L = 7$	18908	0.9
	$L = 8$	73978	0.4
	$L = 9$	276136	0.2
Vase	$L = 7$	37113	1.8
	$L = 8$	135452	0.8
	$L = 9$	432295	0.3

Table 4.2.: Static models characteristics.

About results, we want to start showing compression performances. In particular, we compare compression achieved when working either in RGB or YUV color space for all considered models. Looking at the resulting rate (Fig. 4.4), we can see that YUV is for sure the best choice from this point of view. Moreover, we reported in Fig. 4.5 the same Distortion-Rate function, where rate is now measured in bpv. From this, one can evaluate the compression results independently from n_{occ}. Also looking at this, it is clear that YUV representation is preferable: the compression ratio can be even twice higher when using YUV with respect to RGB, for the same measured distortion.

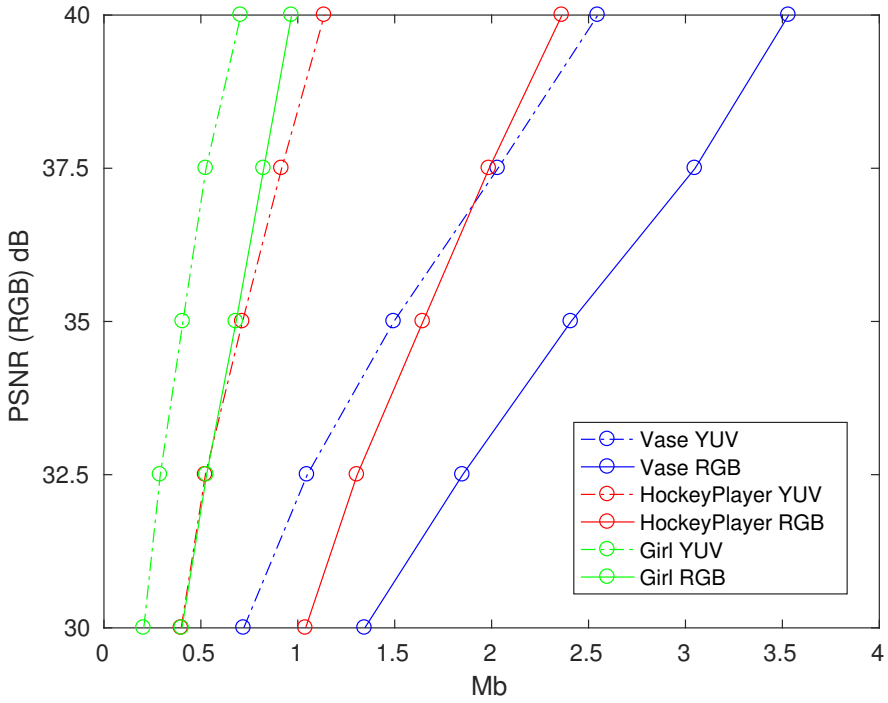


Figure 4.4.: Distortion-Rate plot (rate in Mb). $L = 9$.

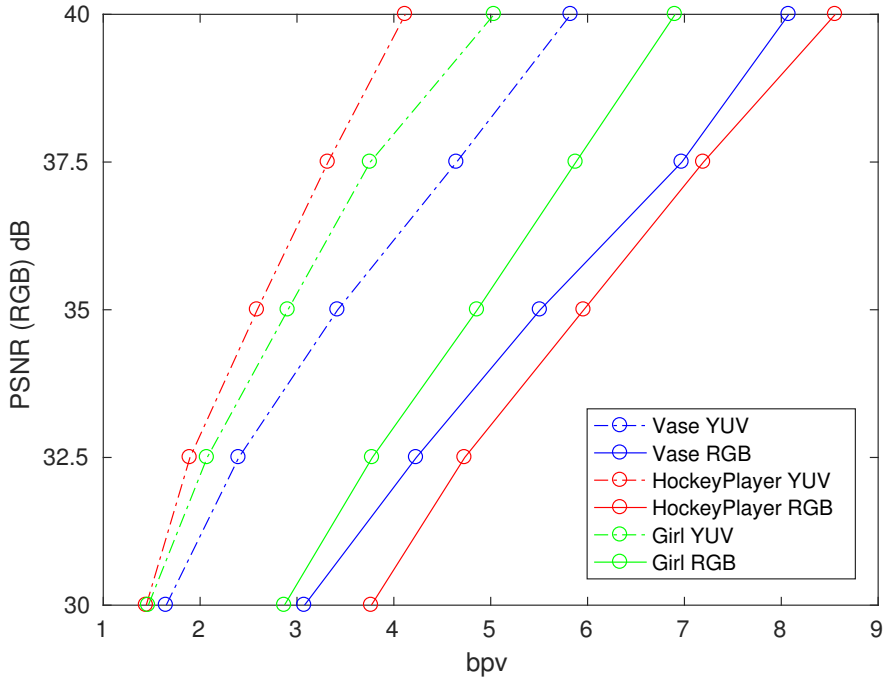


Figure 4.5.: Distortion-Rate plot (rate in bpv). $L = 9$.

We report only results for $L = 9$, because the curves, when using other resolutions are very similar to these ones. In particular, curves where rate is measured in bpv are quite the same for all resolutions. While, of course, when considering the absolute Rate-

Distortion plot for different L values, we find the same shapes reported in Fig. 4.4, but shifted due to very different n_{occ} values.

YUV should be preferred also when considering entropy coding execution times, as can be seen in Table 4.3, where we reported mean entropy coding times for $L = 8$ and $L = 9$ for the same resulting PSNR.

Model	Octree depth	ET (YUV)	ET (RGB)
Girl	$L = 8$	7.8ms	11.0ms
	$L = 9$	25.0ms	30.9ms
HockeyPlayer	$L = 8$	12.0ms	16.2ms
	$L = 9$	52.0ms	96.4ms
Vase	$L = 8$	26.5ms	34.6ms
	$L = 9$	101.5ms	138.1ms

Table 4.3.: Entropy coding execution times. PSNR = 35.0dB.

For these reasons, from now on, all results reported are intended to be obtained working in the YUV color space, even if it is not specified.

In order to compare the complexity of RAHT v1 and RAHT v2 we report results obtained for each model, with respect to different values of L, building a plot where are reported execution time (computed as the average execution time among 30 test) as a function of the resulting PSNR (Fig. 4.6, 4.7 and 4.8).

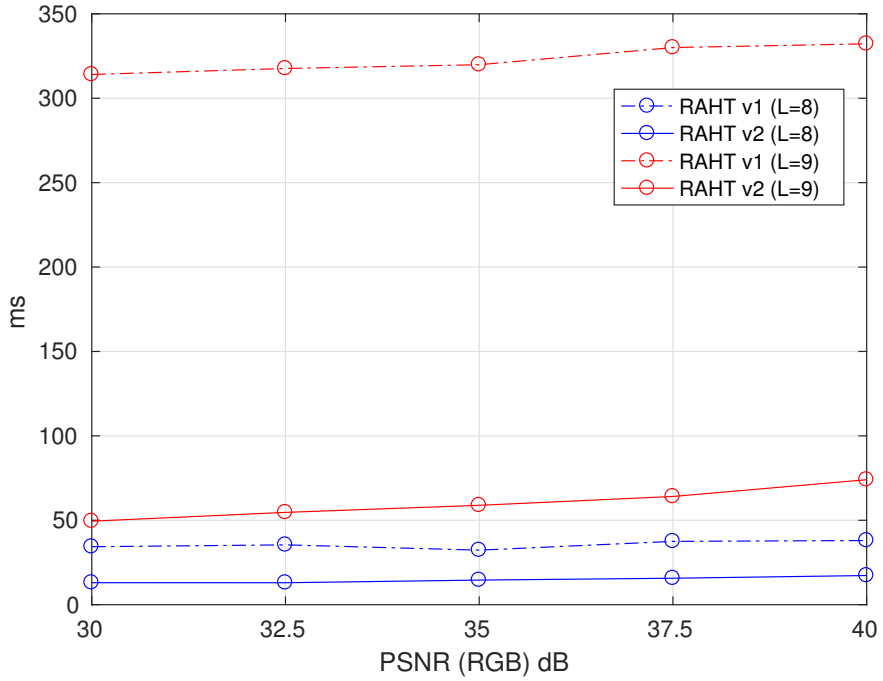


Figure 4.6.: Complexity vs PSNR (model Girl).

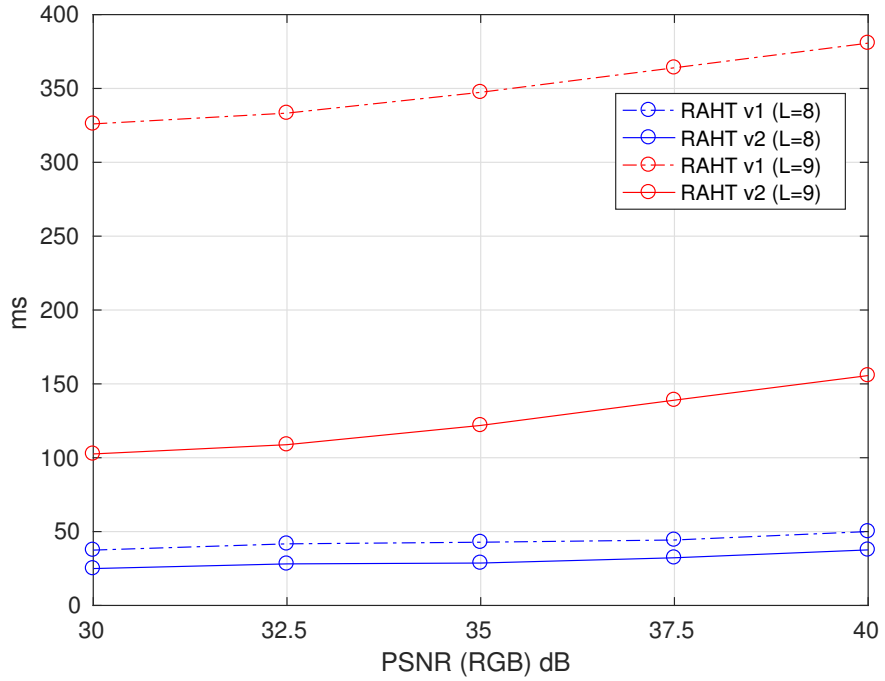


Figure 4.7.: Complexity vs PSNR (model Hockey).

We don't plot $L = 7$ results because, in that case, the whole execution times are in the interval $[7, 11]$ ms for all models and configurations tested: we need denser grids to evaluate performances.

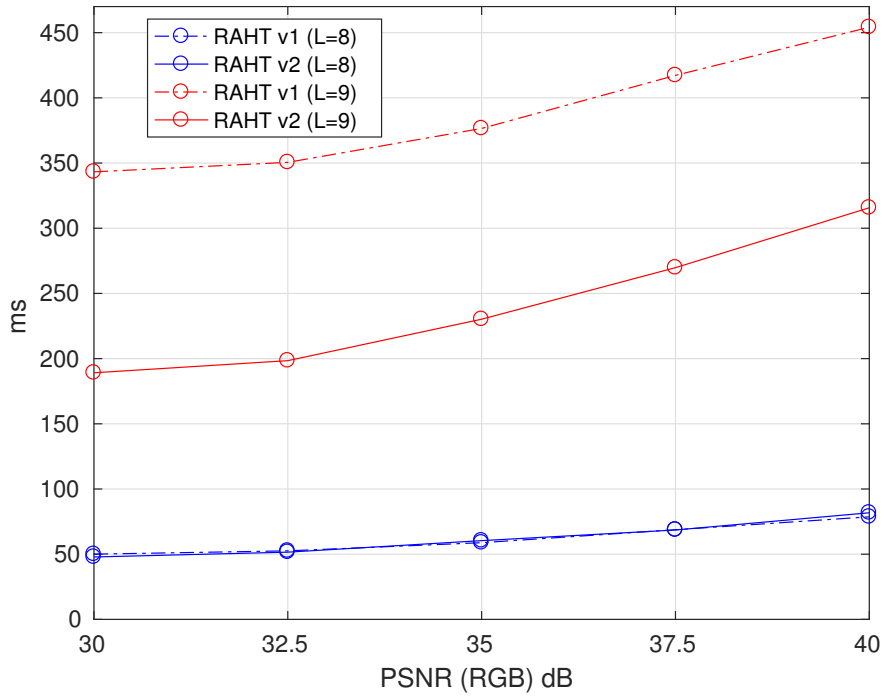


Figure 4.8.: Complexity vs PSNR (model Vase).

From these plots, we can make some general observations:

- the complexity increases with L;
- the complexity increases also with n_{occ}, once fixed L;
- the lower the target distortion, the higher the execution time;
- RAHT v2 is more efficient then RAHT v1, especially for high values of L (only in *Vase* for $L = 8$ their execution time is essentially the same).

For the latter reason, we prefer RAHT v2 for testing our sequence coding system, as anticipated in Chapter 3. But before looking at this, we want to go a little bit in detail and analyze how much time is spent by our system for transform and quantization, and how much time is spent by the entropy coder (Fig. 4.9).

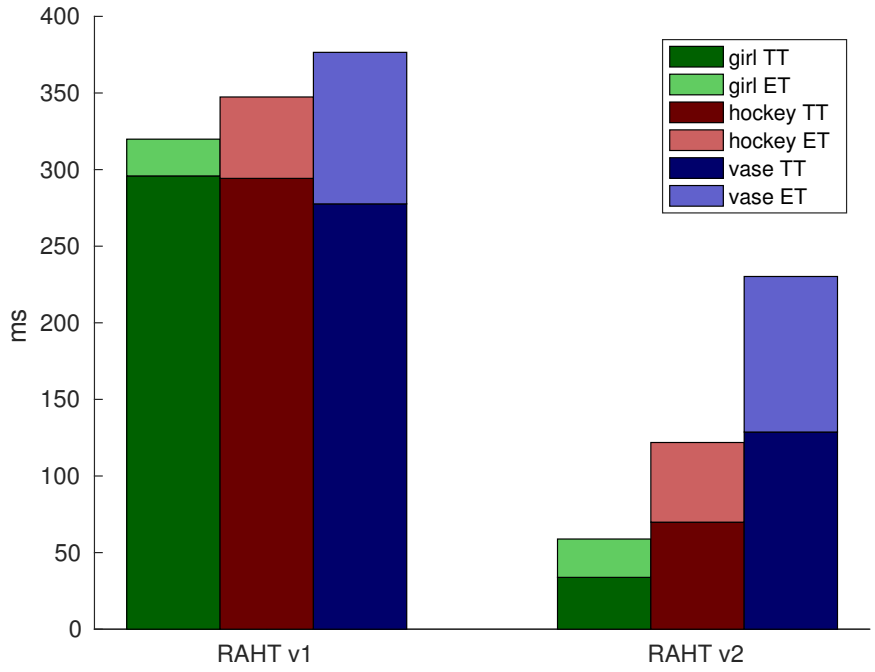


Figure 4.9.: Transform vs Entropy Coding complexity (L = 9, PSNR = 35dB).

As can be seen RAHT v2 is very sensitive to n_{occ}, while RAHT v1 execution time, once fixed L, does not change so much with n_{occ}. This is due to the fact that our second version navigate only inside occupied voxels, while the previous one navigates the entire voxel cube. Finally, we want to notice that also entropy coding execution time is influenced by n_{occ}. One could expect this, in fact, more coefficients needs to be serialized, more time is required to encode the entire data. Moreover, while in RAHT v1 the transform operations dominate the whole execution time, in the second version, they require similar amount of time required for entropy coding.

4.2 Dynamic sequence

To test our dynamic sequence routine, we used the publicly available *Voxelized Upper Body* dataset, provided by Microsoft [21].
It contains five different voxelized 3D video sequences, known as *Andrew*, *David*, *Phil*, *Ricardo* and *Sarah*, captured by 4 frontal RGBD sensors at 30 fps.
Using this dataset, we can test our routine, on data that are very similar to the ones we should handle in a real implementation. In fact, as can be seen in figures below (where we reported the voxelized subjects of each sequence), each voxelized frame presents noisy samples and/or missing surface's data, very common situations in real time acquisitions.

(a) Andrew.

(b) David.

(c) Phil.

(d) Ricardo.

(e) Sarah.

Such dataset is already voxelized and available at a grid resolution of 512^3 or 1024^3 ($L = 9$ or $L = 10$). We haven't considered resolutions higher than 512^3 in this work, so we tested only the lower resolution among the two available. In Table 4.4 we report some meaningful characteristics for each sequence.

Sequence	**Frames**	**Avg.** n_{occ}
Andrew	317	282862,95
David	216	337220,67
Phil	245	331886,97
Ricardo	216	225184,85
Sarah	207	258877,53

Table 4.4.: Video sequences' characteristics.

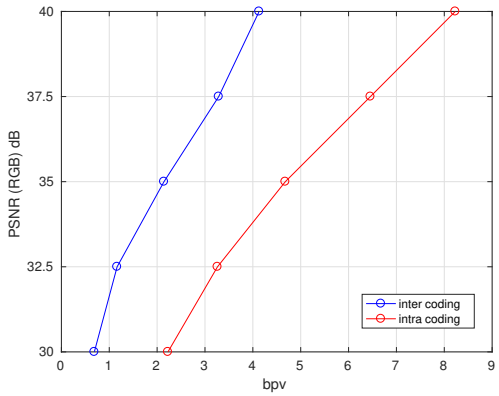
The first thing we want to discuss is the choice of r, the parameter that identifies the neighborhood within which prediction is performed in case no correspondent voxel is found in the previous frame. In order to do this, we report the results obtained encoding the first 50 frames of the sequence *Sarah* for $r = 1$, $r = 2$ and $r = 3$, given a target PSNR of 30dB. We don't report results obtained for all other sequences or configurations because essentially the behavior is exactly the same. Before going on, we want to specify that, from now on, PSNR is intended as the average PSNR of an entire video sequence (i.e. the sum of all PSNRs divided by the number of frames). The same is done for the execution times and rates. Moreover, each sequence encoded using our prediction scheme, if not differently specified, is intended to be composed by a single reference frame (the first one), while all the others are predicted from the one previous one (i.e. GOP is equal to the sequence length).

	PT	**TT**	**ET**	**Rate**	**PSNR**
No prediction		64.5ms	27.6ms	0.67bpv	29.90dB
r=1	19.4ms	62.4ms	18.5ms	0.31bpv	29.91dB
r=1	19.4ms	62.4ms	18.5ms	0.31bpv	29.91dB
r=2	41.4ms	60.4ms	16.2	0.13bpv	29.83dB
r=3	80.8ms	61.7ms	15.3ms	0.08bpv	29.86dB

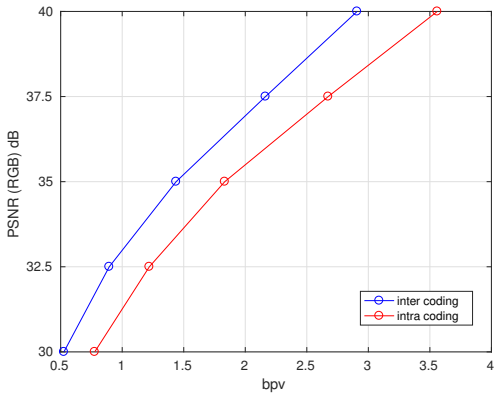
Table 4.5.: Sarah results for different r. Target PSNR: 30dB.

As we can see, each time we increase r of a unit, we double the time required for prediction. On the other side, greater the patch used for prediction, higher the compression achieved. The best trade-off between complexity and compression, in our opinion is when using $r = 2$. In fact, with $r = 1$ compression performances are poor (we have small compression gain w.r.t. coding the sequence without prediction). Instead, when using $r = 3$, we have that $PT > TT + ET$ so, from our point of view, prediction computation becomes too heavy w.r.t. the overall execution time. In the following we will assume $r = 2$ in all the test performed.

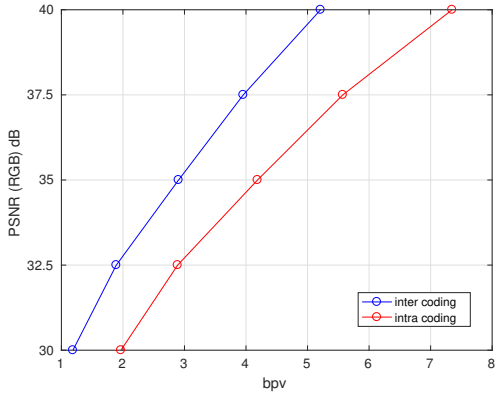
As done for static encoding, we evaluate dynamic sequence compression considering Distortion-Rate plots for each tested sequence (Fig. 4.11). In particular, we are going to compare results obtained coding statically each frame (*intra* coding: no prediction is performed) and the ones obtained using our prediction strategy (*inter* coding).



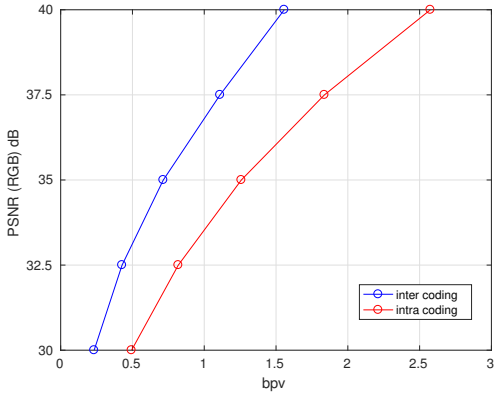
(a) Andrew.



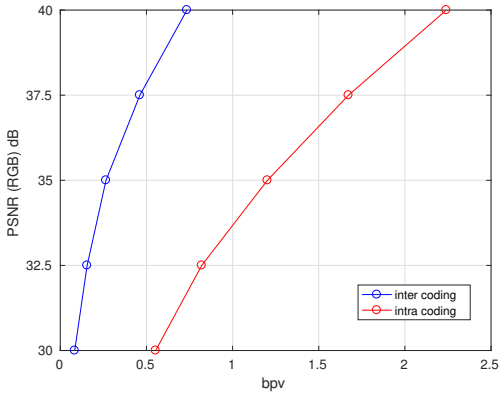
(b) David.



(c) Phil.



(d) Ricardo.



(e) Sarah.

Figure 4.11.: Distortion-Rate plots.

As we can see, our prediction scheme allows improving dynamic coding performances in all tested scenarios. Such improvements depend on the considered sequence, so we decided to report also compression performances for some fixed PSNR values (Table 4.6). Compression performances are expressed as $1 - bpv_{inter}/bpv_{intra}$ where bpv_{inter} is the rate when encoding the frame using prediction, while bpv_{intra} is the same quantity when encoding a sequence without prediction.

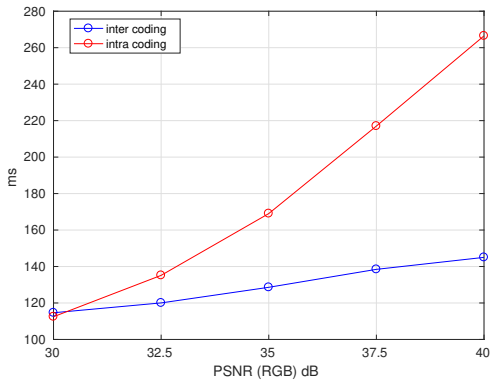
Sequence	30dB	32.5dB	35dB	37.5dB	40dB	Average
Andrew	69.09%	64.30%	54.18%	49.09%	49.80%	57.29%
David	32.22%	26.70%	21.55%	19.05%	18.30%	23.56%
Phil	39.65%	34.51%	30.64%	29.02%	29.08%	32.58%
Ricardo	52.81%	47.80%	43.19%	39.56%	39.52%	44.57%
Sarah	84.88%	81.03%	77.93%	72.39%	67.19%	76.68%

Table 4.6.: Inter coding compression gain.

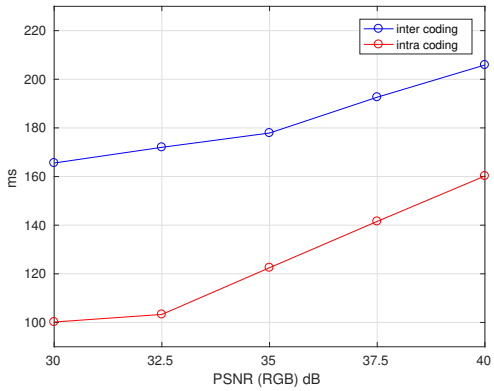
From the reported results we can observe that, using our prediction strategy:

- compression gain depends on the considered sequence;
- compression gain depends also on the target PSNR: the higher the target PSNR, the lower the compression gain;
- on average, in our tests, we save 46.9% of the bits required without prediction.

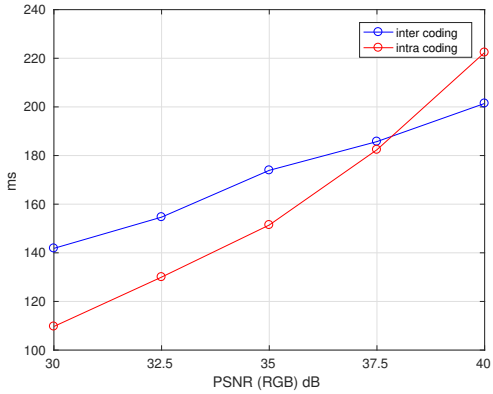
Regarding the execution times, we want to start showing the mean overall execution time for both strategies (i.e. intra and inter coding). As we can see from Fig. 4.12, the complexity in case



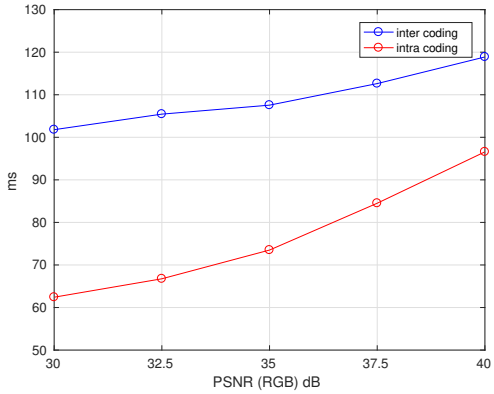
(a) Andrew.



(b) David.

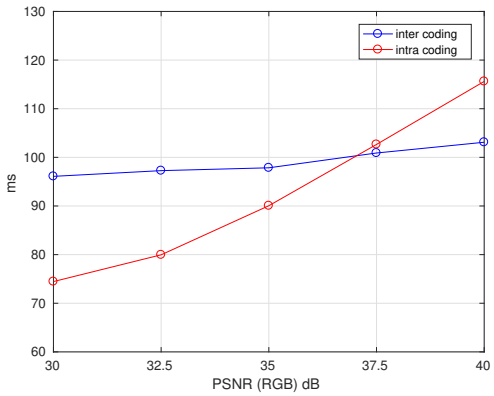


(c) Phil.



(d) Ricardo.

Figure 4.12.: Complexity vs Distortion plots.



(e) Sarah.

Also in this case, the absolute execution time depends on the particular sequence and on the target PSNR. Nevertheless, we can notice some common behaviors:

- for low PSNR, inter coding time is greater than intra coding one, due to PT (required only when predicting);
- when PSNR increases, the difference between inter and intra coding execution time becomes smaller and smaller (intra coding execution time increases faster);
- in some sequences, for the higher PSNR values we have that our prediction scheme is faster than the intra coding one and we think that the same behavior can be observed in all the sequences, further increasing the target PSNR.

This behavior can be better understood looking at Fig. 4.13.

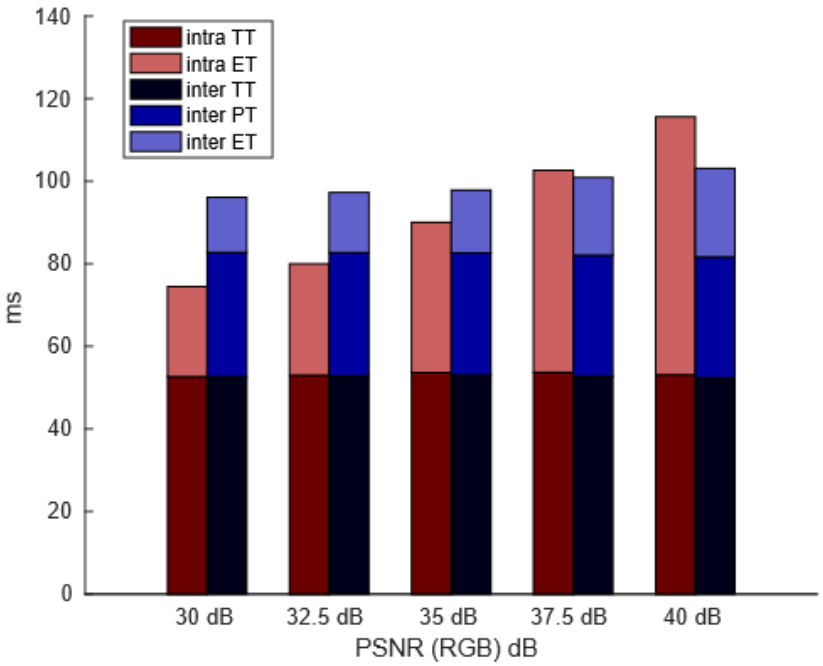


Figure 4.13.: Intra vs inter coding execution time comparison (Sarah).

As expected, TT is the same in both approaches, in fact it depends mainly on frame's geometry that is the same in both situations. Conversely, ET depends on the transformed and quantized coefficients' distribution (besides on n_{occ} as seen in the static case). In fact, it changes with the quantization step or equivalently, with the measured PSNR, as can be seen when looking intra coding execution times: more different coefficients in input at the entropy coder, more time is required for serializing data.
This behavior can be noticed also in inter coding, but the effect is very much smaller. This is because, if prediction works properly, we have that coefficients to be transformed will be close to zero, and the distribution of the entropy coder input will be narrower w.r.t. the one obtained encoding directly the original color attributes. Moreover, for the same reason, inter coding ET is always lower, also for low PSNRs.
PT instead, once fixed r, depends only on the particular video sequence and it's independent w.r.t. the quantization step.
Merging all together, we can conclude saying that while for low PSNRs the gain in ET is masked by PT, for the higher ones inter coding setup could be preferable also from a computational point of view, not only for compression performances.

Finally, we want to conclude this chapter comparing our dynamic sequence encoding system to another recent implementation. In particular, we found that [19] reports results obtained for *Ricardo* using RAHT plus motion compensation for inter coding, so we decided to use this for our comparison. Unfortunately, Distortion-Rate plots are reported considering both geometry and color compression performances all together. Moreover, they used a slightly different metric to compute PSNR: they compute it on YUV color space through:

$$20log_{10}\left(\frac{255}{\sqrt{MSE_Y + 0.35MSE_{geom}}}\right) \tag{4.1}$$

It means that they compute color distortion considering only Y component, while $0.35MSE_{geom}$ takes into account geometry distortion in the overall PSNR, due to the fact that they used a *lossy* approach for geometry too.
So, we recompute Distortion-Rate plot for *Ricardo* using this metric, assuming MSE_{geom} equal to 0, i.e. geometry coded losslessly. Moreover, we used the same number of reference frame within a sequence, setting $GOP = 8$. For geometry rate, we used results obtained in [22], a work developed in parallel to this thesis, focused on lossless geometry compression. Basically, we compared the overall performances of the coding system developed in these two thesis with the one proposed in [19]. The results of inter frame encoding for both approaches are reported in Fig. 4.14. We want to specify that we do not have numerical data for drawing Distortion-Rate plot of [19], but we draw that only inspecting the correspondent plot reported in the paper.
As we can see, using our system we obtain a curve that is $1 \div 1.5$dB higher, given the same rate, for this specific sequence.

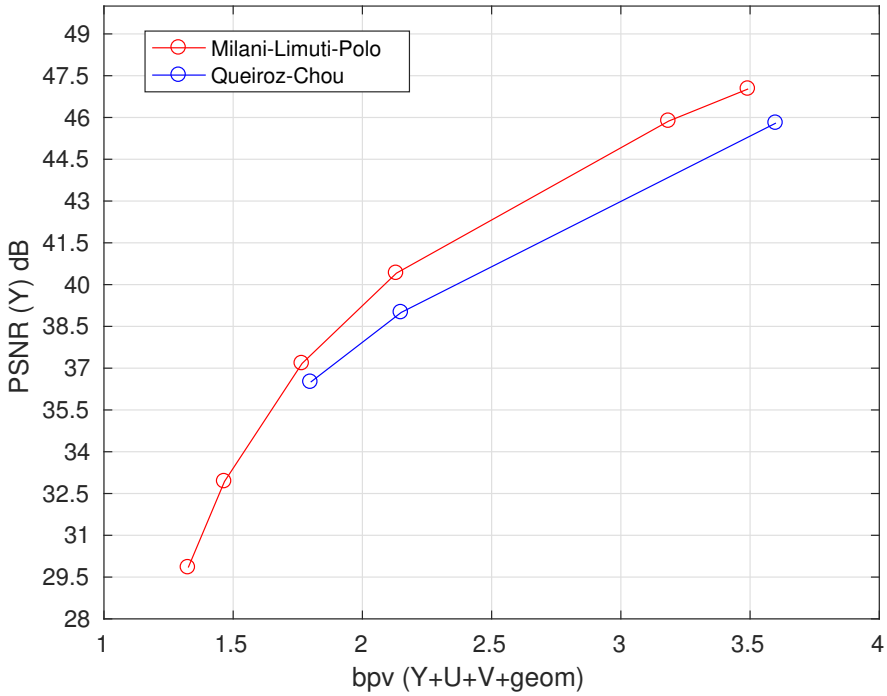


Figure 4.14.: Full encoding system Distortion-Rate comparison (Ricardo).

5

Conclusions

In this work we tested two different RAHT implementations for compressing color attributes within a voxelized point cloud and we propose a novel prediction scheme for encoding voxelized 3D video sequences. In particular, we were interested in evaluating complexity and compression performances of our system. Looking at the results reported in the previous chapter w.r.t. execution times, we can conclude that:

- for resolutions $\leq 128^3$, we satisfy real time constraints (i.e. 33ms execution time) using either RAHT v1 or RAHT v2;
- for 256^3 we satisfy real time constraints (or we are very close to them) using RAHT v2;
- for 512^3 RAHT v2 outperforms RAHT v1 performances, but we are still far away from real time performances, either when encoding a static frame or a video sequence;
- when encoding a video sequence, prediction increases the complexity of the system for low target distortions, while for higher ones it should be preferred from a computational point of view (w.r.t. intra coding);

For these reasons, we think that our code should be further optimized in order to deal with high resolution models satisfying real time constraints. Such optimization can be obtained, in our opinion, by parallelizing the tasks to be executed (e.g. computing new coordinates for the next decomposition steps in parallel when performing the transform or entropy coding AC coefficients after each decomposition iteration, instead of waiting the transform to be done before serializing them). Hence, an implementation that runs on GPU should be considered to improve parallelization performances. One can think also about dividing the voxel grid into eight sub-volumes with halved dimensions (or even smaller sub-volumes) and perform the transformation in parallel on each cube.
Regarding compression:

- compression in YUV color space outperforms results obtained using RGB coordinates;
- when encoding video sequences, inter frame coding must be preferred, allowing to save about half the bit required when performing intra coding;
- effective compression results are highly dependent on the particular 3D model (e.g. 0.08bpv for sequence *Sarah* at 30dB but 1.18bpv for *Phil*, at the same distortion);

- also our prediction scheme performances depends on the specific sequence (e.g. it provides an average gain of 23.6% for *David* and 76.7% for *Sarah*).

Further improvements should be obtained, for example, by using different entropy coders for AC generated at a certain decomposition level, instead of encoding all coefficients for a certain color component with the same coder. Moreover, when encoding sequences, one can try to initialize the arithmetic coder using the coefficients probability distribution of the previous frame. Also a better prediction strategy can be considered, if it is feasible with real-time constraints. Finally, we want to report two useful features that could be added to our system, such as:

- progressive color decoding, to allow decoder to partially decode color information, like it is for geometry;
- color residual encoding, in order to restore original point cloud color attributes.

A

Appendix 1:Arithmetic Coder

Arithmetic coding is a method for entropy coding data generating variable-length codes. It is especially useful when dealing with sources with small alphabets or with highly skewed probabilities. Before describing how it works let us introduce some notation and assumptions:

- $\mathcal{A} = \{a_1, ..., a_m\}$ is the alphabet (i.e. the set of all possible values a source can generate);
- $X(a_i) = i$ is a random variable defined on $\mathcal{A}$;
- $P(X = i) = P(a_i)$ is the probability of the i-th symbol to be generated by the source;
- symbols are assumed to be independent and identically distributed (iid);
- N is the total number of symbols generated by the source (i.e. we want to encode);
- $p(X^N) = P(a_1, ..., a_N) = P(a_1)P(a_2)...P(a_N)$ is the probability for a certain sequence to be generated;
- $F_X(i) = \sum_{k=1}^{i} P(X = i)$ is the cumulative distribution function (cdf) of the source;

Given this, we can notice that, if $P(a_i) \neq 0, \forall i \in 1, ..., m$, then we have that $F_X(i) \neq F_X(j), \forall i \neq j$. Arithmetic coder exploits this simple observation in order encode a sequence of symbols, relating $p(X^N)$ to a specific interval in the region $[0, 1) \subset \mathbb{R}$. In fact, arithmetic coding procedure acts as follows:

1) partition $[0, 1)$ in m disjoint intervals of the form $[F_X(i-1), F_X(i))$;
2) read the symbol a_k generated by the source;
3) restrict the interval representing the encoded sequence to $[F_X(k-1), F_X(k))$;
4) divide $[F_X(k-1), F_X(k))$ in m disjoint interval like in 1);
5) repeat 2),3) and 4) until the sequence is over;
6) assign a binary tag to the resulting interval and transmit it.

This process is illustrated in Fig. A.1 for a source with $\mathcal{A} = \{a, b, c\}$ and $P(a) = 0.5$, $P(b) = 0.3$ and $P(c) = 0.2$; and where we assume we want to encode the sequence "aba".

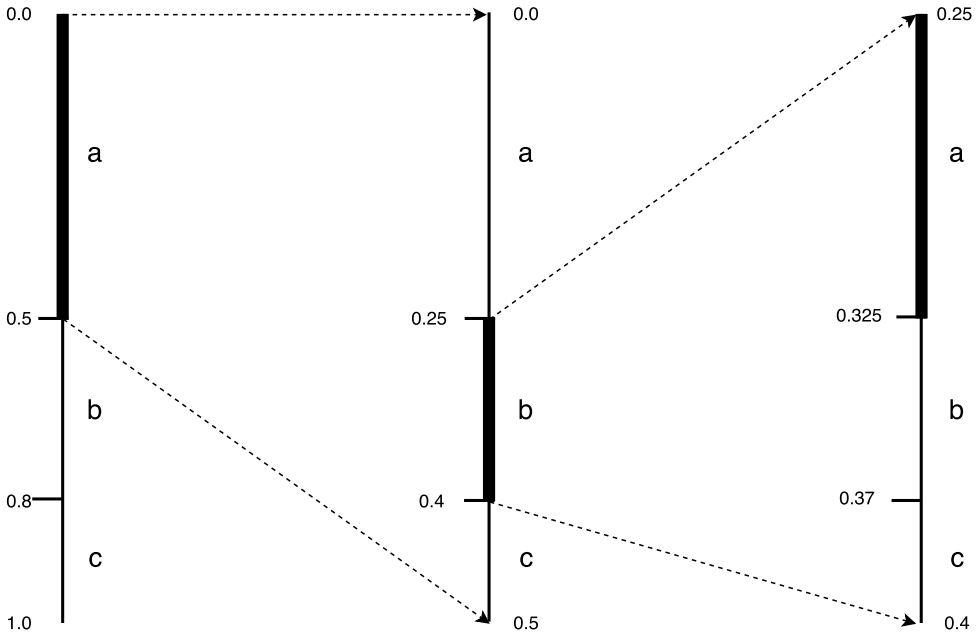


Figure A.1.: Arithmetic encoding example.

In this simple example, the interval representing the specific sequence is $[0.25, 0.325)$. What is left is how to assign a tag to a certain interval. The tag can be the binary representation of any point within the interval, but it is a common behavior to select its midpoint (or more precisely only its decimal part, the integer one is always 0). In theory, one should have infinite bits to represent a specific point $\in \mathbb{R}$ and ensure unique decodability. However, can be shown that it sufficient to represent it using $l + 1$ bits, where:

$$l = \left\lceil -log_2 \left(p \left(X^N \right) \right) \right\rceil$$

in order to guarantee this property (i.e. in order to have a useful code).
In our example, the midpoint is $\bar{I} = 0.2875$, while $l = 4$; so 5 bits are required to encode this sequence and the correspondent tag is 01001.
At the decoder side, the bit stream is received and converted to the decimal tag, obtaining the value 0.2815 (remember that the tag is a truncated representation of $\bar{I}$).
Then, the process is similar to the encoding one:

1) divide the unit interval as explained before;

2) find the sub-interval that contains the tag, let us call it k, and decode the first symbol (a_k in this case);

3) restrict the considered interval to $[F_X(k-1), F_X(k))$;

4) divide it according to the symbol probability distribution;

5) repeat 2),3) and 4) until the sequence is over.

So, referring to our example and looking at Fig. A.1, we can see that at the first iteration the decoded tag belongs to $[0, 0.5)$ that corresponds to "a". At the second one it is in

$[0.25, 0.4)$, so the second decoded value is "b" and so on, finally obtaining the original sequence.
This procedure assumes that $P(a_i)$ are known both by encoder and decoder. In practical applications that is not true in general. For this reason, it possible to modify the procedure already presented, allowing the encoder and decoder to learn the alphabet probability distribution, while encoding (or decoding) a certain sequence. In order to do this, the procedure starts assigning equal probability to all symbols $P(a_i) = 1/m$ $\forall a_i \in \mathcal{A}$. Then, after encoding (or decoding) each symbol, the routine updates the probability distribution according to the effective symbol frequency.
This procedure defines the so called *adaptive arithmetic coder*. To improve the performances of this system, it is also possible to initialize the symbol probability distribution to some known distribution (e.g. Gaussian or Laplacian), according to the one we expect from our source, and then make the procedure adapt starting from this.

Bibliography

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

[2] S. Milani, "Fast point cloud compression via reversible cellular automata block transform," *ICIP 2017*, 2017.

[3] D. C. Garcia and R. L. de Queiroz, "Context-based octree coding for point-cloud video presentation," 2017. [Online]. Available: http://sigport.org/1812

[4] C. Zhang, D. A. F. FlorÃłncio, and C. T. Loop, "Point cloud attribute compression with graph transform." in *ICIP*. IEEE, 2014, pp. 2066–2070. [Online]. Available: http://dblp.uni-trier.de/db/conf/icip/icip2014.html#ZhangFL14

[5] R. L. de Queiroz and P. A. Chou, "Compression of 3d point clouds using a region-adaptive hierarchical transform." *IEEE Trans. Image Processing*, vol. 25, no. 8, pp. 3947–3956, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/tip/tip25.html#QueirozC16

[6] E. Pavez, P. A. Chou, R. L. de Queiroz, and A. Ortega, "Dynamic polygon cloud compression," *CoRR*, vol. abs/1610.00402, 2016.

[7] P. Milgram and F. Kishino, "A taxonomy of mixed reality visual displays," *IEICE Trans. Information Systems*, vol. E77-D, no. 12, pp. 1321–1329, Dec. 1994. [Online]. Available: http://vered.rose.utoronto.ca/people/paul_dir/IEICE94/ieice.html

[8] R. T. Azuma, "A survey of augmented reality," *Presence: Teleoper. Virtual Environ.*, vol. 6, no. 4, pp. 355–385, Aug. 1997. [Online]. Available: http://dx.doi.org/10.1162/pres.1997.6.4.355

[9] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre, "Recent advances in augmented reality," *IEEE Comput. Graph. Appl.*, vol. 21, no. 6, pp. 34–47, Nov. 2001. [Online]. Available: http://dx.doi.org/10.1109/38.963459

[10] M. Mekni and A. Lemieux, "Augmented reality: Applications, challenges and future trends," 2014.

[11] S. Orts-Escolano, C. Rhemann, S. Fanello, W. Chang, A. Kowdle, Y. Degtyarev, D. Kim, P. L. Davidson, S. Khamis, M. Dou, V. Tankovich, C. Loop, Q. Cai, P. A. Chou, S. Mennicken, J. Valentin, V. Pradeep, S. Wang, S. B. Kang, P. Kohli, Y. Lutchyn, C. Keskin, and S. Izadi, "Holoportation: Virtual 3d teleportation in real-time," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. New York, NY, USA: ACM, 2016, pp. 741–754. [Online]. Available: http://doi.acm.org/10.1145/2984511.2984517

[12] O. Bimber, R. Raskar, and M. Inami, "Spatial augmented reality," *AK Peters Wellesley*, 2005.

[13] F. Zhou, H. B.-L. Duh, and M. Billinghurst, "Trends in augmented reality tracking, interaction and display: A review of ten years of ismar," in *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, ser. ISMAR '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 193–202. [Online]. Available: http://dx.doi.org/10.1109/ISMAR.2008.4637362

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

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

[16] R. Schnabel and R. Klein, "Octree-based point-cloud compression," in *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, ser. SPBG'06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 111–121. [Online]. Available: http://dx.doi.org/10.2312/SPBG/SPBG06/111-120

[17] R. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 1 –4. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5980567&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D5980567

[18] 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.

[19] R. L. de Queiroz and P. A. Chou, "Motion-compensated compression of dynamic voxelized point clouds," *IEEE Transactions on Image Processing*, vol. 26, no. 8, pp. 3886–3895, Aug 2017.

[20] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987. [Online]. Available: http://doi.acm.org/10.1145/214762.214771

[21] "Voxelized upper bodies dataset." [Online]. Available: https://jpeg.org/plenodb/pc/microsoft/

[22] S. Limuti, "Compression of 3d models for augmented reality," 2016/2017.