DISSERTAÇÃO DE GRADUAÇÃO

Point Cloud Compression: Silhouette 3D: Migration from Matlab to C++

Otho Teixeira Komatsu

Graduação em Engenharia de Computação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA/CIÊNCIA DA COMPUTAÇÃO
INSTITUTO DE CIÊNCIAS EXATAS
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Instituto de Ciências Exatas

DISSERTAÇÃO DE GRADUAÇÃO

Point Cloud Compression: Silhouette 3D: Migration from Matlab to C++

Otho Teixeira Komatsu

Dissertação de Graduação submetida ao Departamento de Ciência da Computação como requisito parcial para obtenção do grau de Graduação em Engenharia de Computação

FICHA CATALOGRÁFICA

T. KOMATSU, OTHO
Point Cloud Compression:Silhouette 3D: Migration from Matlab to C++ [Distrito Federal] 2021.
xvi, 66 p., 210 x 297 mm (CIC/IE/UnB, Graduação, Engenharia de Computação, 2021).
Dissertação de Graduação - Universidade de Brasília, Instituto de Ciências Exatas.
Departamento de Ciência da computação

1. Point Clouds	2. Signal Compression
3. Geometry Compression	4. Intra frame compression
I. CIC/IE/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

T.KOMATSU, OTHO (2021). *Point Cloud Compression:Silhouette 3D: Migration from Matlab to C++*. Dissertação de Graduação, Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, 66 p.

CESSÃO DE DIREITOS

AUTOR: Otho Teixeira Komatsu
TÍTULO: Point Cloud Compression:Silhouette 3D: Migration from Matlab to C++.
GRAU: Graduação em Engenharia de Computação   ANO: 2021

Otho Teixeira Komatsu
Depto. de Ciência da Computação (CIC)
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

Dedico esse trabalho aos meus pais, Kazuhiko e Tânia, por todo o zelo e carinho na minha educação, cuidado e crescimento como pessoa, criação graças a qual me pertime escrever o presente trabalho.

AGRADECIMENTOS

A Deus dou graças pela oportunidade do presente trabalho, pela minha saúde e condições de poder realizar minhas atividades na produção do presente trabalho no contexto da pandemia.

Agradeço aos professores Eduardo Peixoto e José Edil Guimarães por confiarem nas minhas capacitações para me cooptarem no projeto, e darem todo o suporte possível de forma comprometida, amistosa e descontraída para a realização do presente trabalho de graduação. Agradeço à equipe do projeto e ao parceiro de TCC pela colaboração e comprometimento para que o projeto fosse concluído.

Agradeço a todos os colegas e amigos da UnB que fizeram parte da minha tragetória desses ultimos 5 anos de curso, assim como amizades e aprendizados na empresa júnior Struct. Cada momento fez parte do meu crescimento como pessoa e da minha história nessa fase de graduação, desde os momentos em que precisei de ajuda até aqueles em que tivemos conquistas.

Agradeço aos professores e monitores da UnB que fizeram parte da minha formação e me confirmaram minha paixão em estudar e trabalhar na área de engenharia de computação.

Por fim, agradeço à minha família, amigos e à igreja por todo o carinho e acompanhamento durante o período da realização desse projeto, os quais me confortam por fazerem parte da minha vida. Obrigado a todos.

RESUMO

Com o aumento des pesquisas em métodos de compressão point clouds, uma variedade de algoritmos foram desenvolvidos adotando uma particular abordagem com o objetivo de gerar codificadores mais perfomáticos, isto é, com melhores taxas de compressão. Um dos algoritmos desenvolvidos trata-se do Silhouette 3D (S3D), criado pelo trabalho do Peixoto, que utiliza-se de técnicas de compressão de imagens. A versão inicial feita em Matlab apresenta resultados superiores comparado à maioria dos encoders de *point cloud*, inclusive o GPCC codec; indicando a eficiência do método. Para a submissão do algoritmo ao MPEG para considerarem a sua proposta, o codificador deve ser migrado para uma linguagem mais comum a *codecs*, além de aprimorar sua performance utilizando-se de linguagem compilada. Assim, uma equipe foi formada para iniciar o processo de migração do código original Matlab para C++, em que cada membro foi responśavel por uma parte específica do processo. Esse trabalho explana tanto o processo de modelagem quanto implementação das principais classes e abstrações de dados do algoritmo S3D, além disso avalia sua performance considerando sua taxa de compressão e tempo de execução, cujo valor pode ser reduzido. Os resultados mostram que a nova versão migrada apresenta uma taxa de compressão superior assimilar à versão original, e um significativa redução do seu tempo de execução.

ABSTRACT

With the increase on point cloud compression methods research, a variety of algorithms was developed with a particular approach in order to propose a more performative encoder, i.e, a better compression rate. One of these algorithms is the Silhouette 3D(S3D), developed by Peixoto's work, using image compression techniques on the process. The initial version implemented in Matlab presented outperforming results compared to the most encoders and the GPCC codec, indicating efficiency of the method. In order to submit this algorithm to MPEG for consideration, the encoder had to be migrated to a more usual language to codecs and improving its performance using a compiled language. Thus, a team was formed to start the process of migrate the original Matlab implementation to C++, where each member was responsible for a specific part of this process. This work explains the process of modelling and implementing the main classes and data abstractions from the S3D algorithm, and assess its performance considering its compression rate and execution time, which value should be reduced. The results shows that the new migrated version presents a similar compression rate as the original, and a significantly reduction on the algorithm's time execution.

CONTENTS

1 INTRODUCTION 1
 1.1 CONTEXT 1
 1.2 MOTIVATION 2
 1.3 PROBLEM 7
 1.4 PURPOSE 7

2 LITERATURE REVIEW 9
 2.1 POINT CLOUDS 9
 2.2 SIGNAL COMPRESSION 10
  2.2.1 GEOMETRY COMPRESSION 13
 2.3 CABAC 13
  2.3.1 ARITHMETIC CODING 14
  2.3.2 CONTEXT ARITHMETIC CODING 18
  2.3.3 CONTEXT ADAPTIVE ARITHMETIC CODING 18
  2.3.4 CONTEXT ADAPTIVE BINARY ARITHMETIC CODING 19
 2.4 SILHOUETTE 3D 21
  2.4.1 SILHOUETTE 22
  2.4.2 SILHOUETTE TREE 22
  2.4.3 SILHOUETTE DECOMPOSITION 22
  2.4.4 ENCODING 26
  2.4.5 DECODING 27

3 IMPLEMENTED SOLUTION 30
 3.1 MATLAB PROJECT 30
 3.2 PROJECT STRUCTURE 31
 3.3 C++ MODELING 33
  3.3.1 PIXEL AND VOXEL 33
  3.3.2 IIMAGE 34
  3.3.3 IMAGESPARSE 35
  3.3.4 IMAGERASTER 37
  3.3.5 POINTCLOUD 37
 3.4 DOCUMENTATION 39

4 RESULTS 42
 4.1 GATHERED RESULTS 42

5 CONCLUSIONS 53

6 APPENDIX A 54

6.1 ARITHMETIC CODING 54
6.2 CONTEXT ARITHMETIC CODING 56
6.3 CONTEXT ADAPTIVE BINARY ARITHMETIC CODING 59

REFERENCES 65

FIGURE LIST

1.1 3d Mesh models 1
1.2 St. Gallen Cathedral point cloud example 3
1.3 Marketplace Feldkirch point cloud example 4
1.4 Dyadic decomposition on a single point cloud ilustrated 5
1.5 S4D decomposition process and context gathering 6

2.1 Point clouds categories charts 10
2.2 The octree geometry compression illustration 14
2.3 Arithmetic coding tag's interval generating 16
2.4 Ricardo9 point cloud projections 23
2.6 A silhouette decomposition ilustration 25
2.7 A silhouette decomposition example with 3 levels 26
2.8 Contexts used to encode a pixel 27
2.9 The binary tree node's encoding process using contexts. 28
2.10 Slices images merged 29

3.1 S3D data structures classes and it's relations 34
3.2 The IImage, ImageSparse and ImageRaster relationship 35
3.3 Point Clouds Class methods 38
3.4 Comment documentation on Visual Studio Code 39
3.5 Doxygen HTML documentation generated 40
3.6 Doxygen's classes inheritance 40

4.1 All point clouds used as input to assess the C++ S3D performance 47
4.2 Andrew compression rate per frame. 48
4.3 David compression rate per frame. 49
4.4 Phil compression rate per frame. 50
4.5 Ricardo compression rate per frame. 51
4.6 Sarah compression rate per frame. 52

TABLES LIST

2.1 Table after encoding a_1 19

2.2 Table after encoding all symbols 19

4.1 Compression algorithms rates comparisons: 43

4.2 Matlab and C++ S3D implementation comparisons 43

4.3 C++ migration S3D gains over original implementation 44

4.4 Encoding implementation profiling 44

4.5 Encoding implementation profiling 44

4.6 Decoding implementation profiling 45

4.7 Decoding implementation profiling 45

6.1 Example 1 - Symbols probability table 54

6.2 Example with contexts - Symbols context probability table 56

1 INTRODUCTION

1.1 CONTEXT

Technology improvements based on three dimensional (3D) objects scanning and detecting from special cameras is a history landmark that is a trend on the industry recently. With the development of more consumer accessible devices, 3D models application is not restricted only on research, but at industry and consumers market. We could cite: virtual reality (VR) broadcasting, 3D videos, robotics, autonomous navigation based on 3D large-scale dynamic maps, geographic information system (GIS) applications, tele-immersive applications [1], as at work [2], animations, gaming e scientific visualization applications [1]. In the paper [3] we can observe a montain rendering example.

An interesting example is free viewpoint views and videos, where active people and objects being captured and conveyed in real time to remote locations can be seen as 3D motion data in multiples positions and angles. Furthermore, this 3D objects rendering to remote locations allows collaboration as if all parties were co-located. All this captures are made by multiple cameras (infrared and regular) arrangement, where the real time capturing and rendering are made possible by using powerful graphics processing units (GPUs). Finally, the rendering are visible through special render glasses [4]. Because of the research development on this area, and its growing consumption since its introduction in the market due to recent discoveries, there was an emergence of alternative methods for capturing and representing the data of these captured geometric volumes. Each of these developed methods seek to solve problems related to predecessor standard and optimize the representations, along with the development of hardware power.

With recent advances in 3D sensing and high-performance computing, point clouds gained more attention [1]. For example, mobile devices, such as Apple's iPhone X, and Sony's Xperia XZ1, have supported point-cloud representation at the level of a few hundred thousand points. However, along with point cloud technology advances and improvements, permitting representation of well defined complex objects, a bottleneck increase. Overall, the point clouds raw data representation occupies huge amount of memory storing size. And this intensifies mainly because of the more demanding larger representations, as we can

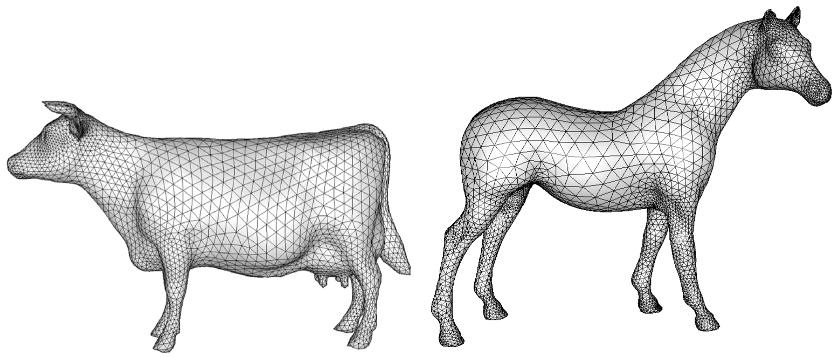

(a) The Cow mesh model (b) The Horse mesh model

Figure 1.1: 3d Mesh models.

observe in figure 1.2, 1.3 [5]. With high-qualities models, more data are stored with. Geometry (meaning more points), color, normal, reflectance, and other attributes [1]. Considering that many systems and applications demands on using these large and high-quality point clouds, by storing, transmissing, processing and rendering [6], specially on real-time applications, it is of utmost importance to efficiently compress the data in order to perform those process. Consequently, the development of compression technologies for point cloud has been a field of intensive investigation in the research community [1].

One of the main concerns is decoding efficiency, ensuring that end users have a coding scheme whose compressed models reconstruction time is fairly good. This also requires that the decoders are simple to implement. Another important concern is the memory usage on the codec as an important parameter, providing a high compression ratio [6]. The technique of 3D model coding has been studied for more than a decade. Previously, 3D mesh compression was a main topic in graphics compression. But with the point cloud advance, point cloud data compression has been a recently an intensive research area. One of the coding algorithms developed that we could cite is the compressors at the works [7], [6], [4]. In 2014, MPEG began an exploration activity on *Point Cloud Compression* (PCC). In 2017, MPEG made a *call for proposals* (CfP) for PCC in order to attend the fast-growing interest of point-cloud-based applications industry. From that, a set 13 technical proposals was evaluated in October 2017. As an outcome, three different technologies were chosen as test models for three disparate content categories [1]. Since then, a great number of papers and methods proposals for point cloud compression has been studied last years, as this area research becomes more important on industry and academically.

1.2 MOTIVATION

Among the researches and papers published proposing point cloud compression techniques, in 2019 a paper published at IEEE VCIP by the professor Eduardo Peixoto and Rodrigo Rosário proposed an Intra-frame geometry compression method based on JBIG and an important technique based on Boolean Decomposition [8]. Following this paper, another technique was developed inspired on the Boolean Decomposition from the previous work, published on 2020 by Peixoto [9], proposing a Lossless Intra-Frame Compression of Point Cloud Geometry Using Dyadic Decomposition. In this work, the technique proposed a recursive splitting of the father point cloud in children halves, and for each point cloud child half a silhouette from the volume occupied is obtained on a given axis. Using the father and children's silhouettes, a similar decomposition to the previously proposed Boolean Decomposition is made as in figure 1.4 [10], but now based on Silhouette Decomposition. Additionally, the silhouette decomposition encoding is made by a Context Adaptive Binary Arithmetic Coder and a single mode option on the dyadic decomposition process. This technique is called **Silhouette-3D** (**S3D**). Later on literature review section, it will be explained more clearly.

Comparing the compression rate, *bits per occupied voxel* (bpov), analysed on two public databases, the **S3D** outperforms all *state-of-art* intra coders tested, including MPEG G-PCC TMC13 v7.0. Futhermore, it outperforms a recent *state-of-art* inter coder Parent Node Inheritance plus Super-Resolution P (Full) [9]. Another approach was developed using lossy approach, using *downsampling* and skipping a certain number of consecutive slices. Similarly, it outpeforms TMC13 v7.0 *Trisoup* and *Octree* for bitrates greater than 0.3

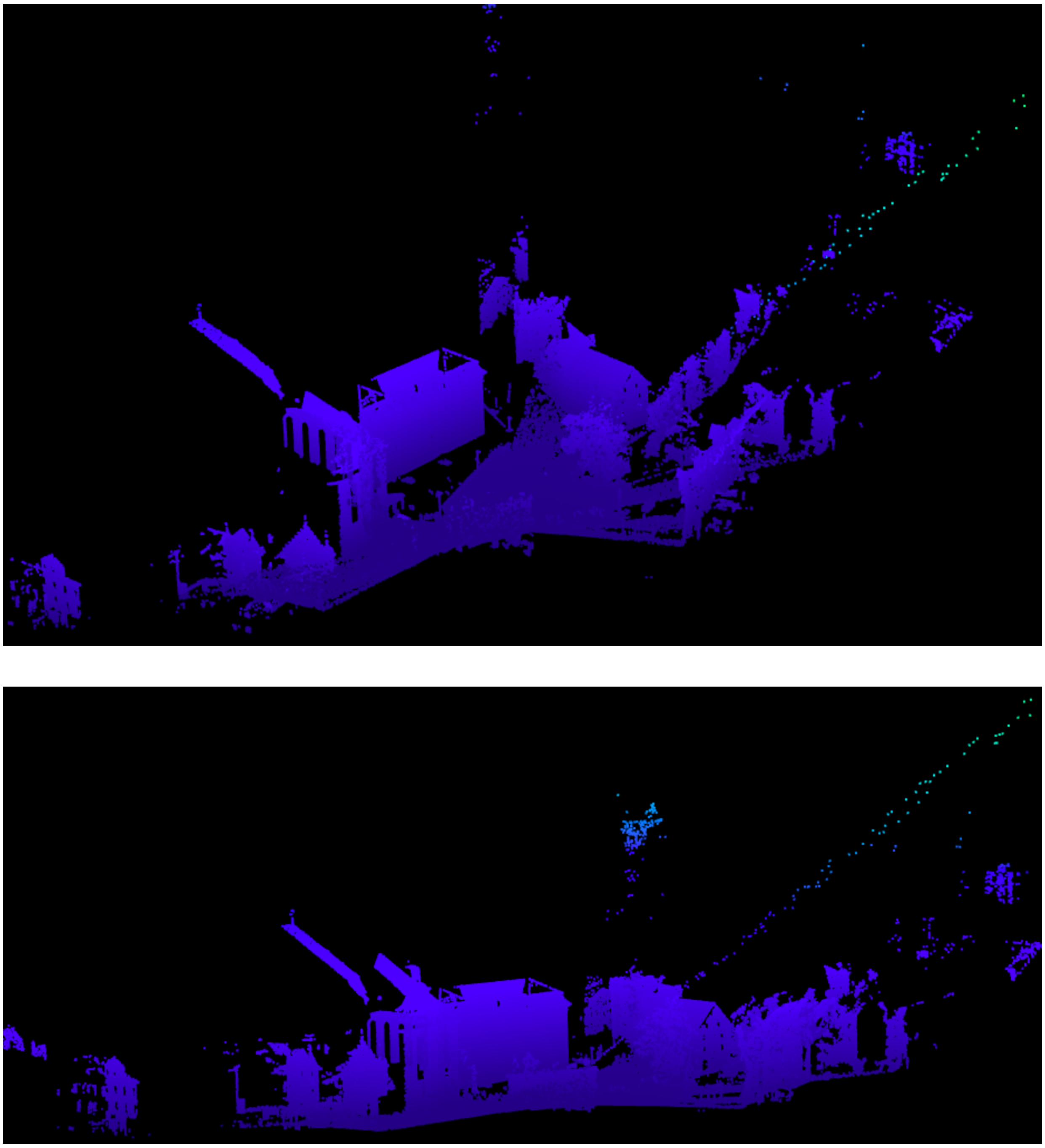


Figure 1.2: Large point cloud example 1 - St. Gallen Cathedral point cloud 3d model (32.342.450 points).

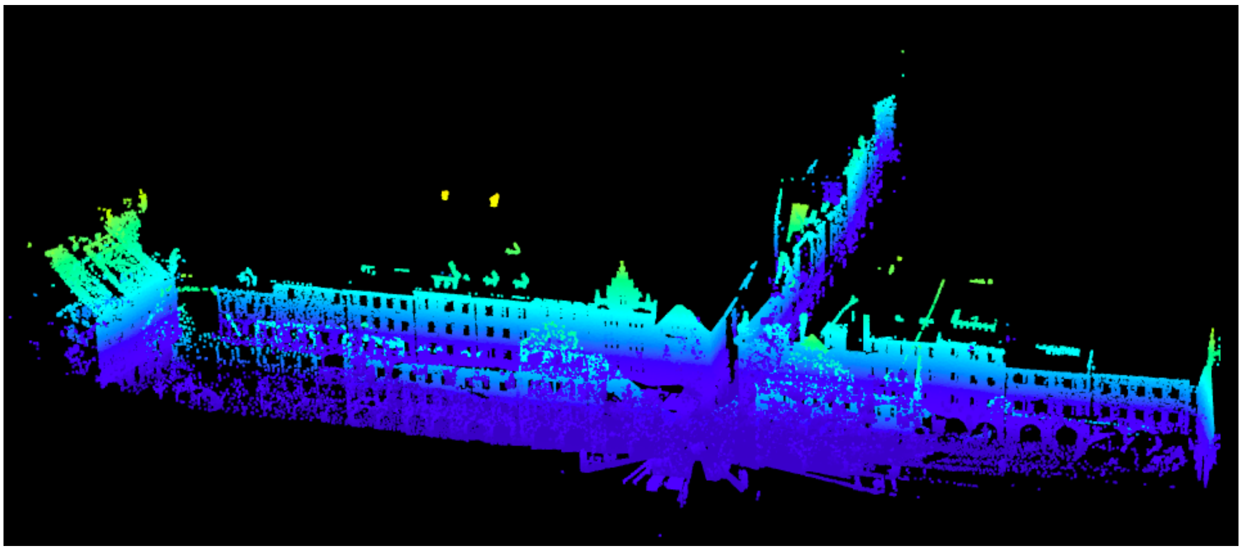
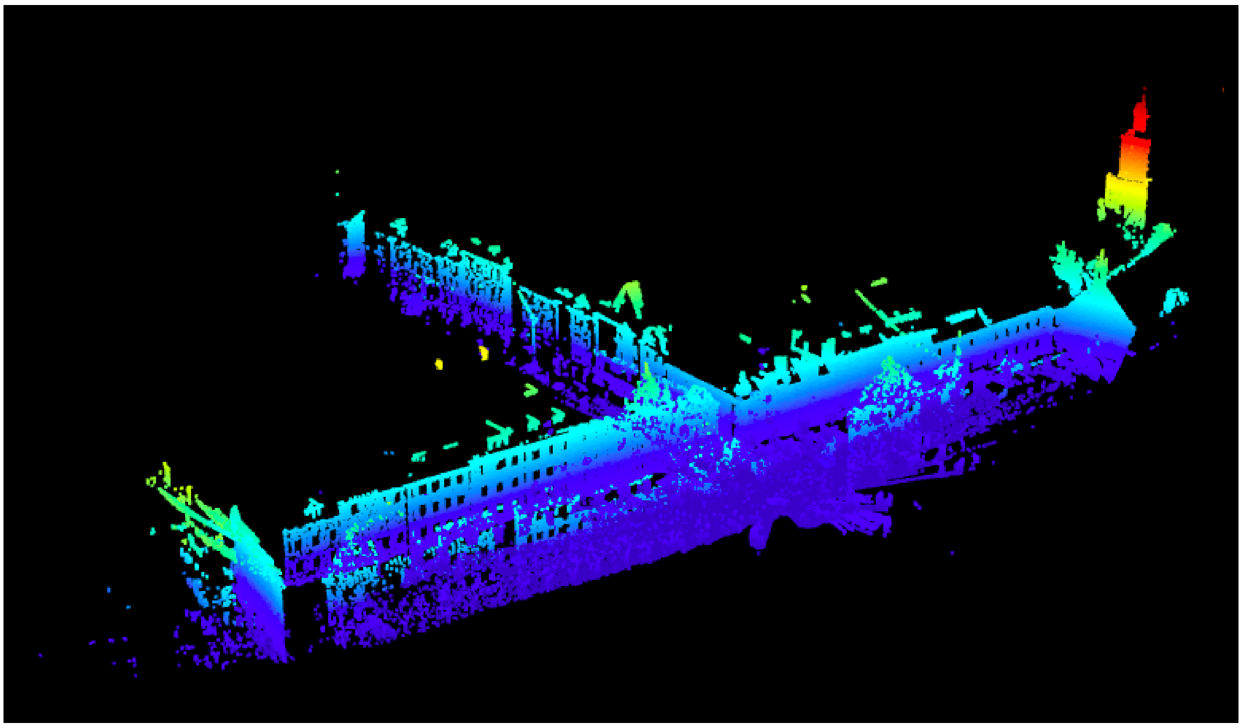



Figure 1.3: Large point cloud example 2 - Marketplace Feldkirch point cloud 3d model (22.760.334 points)

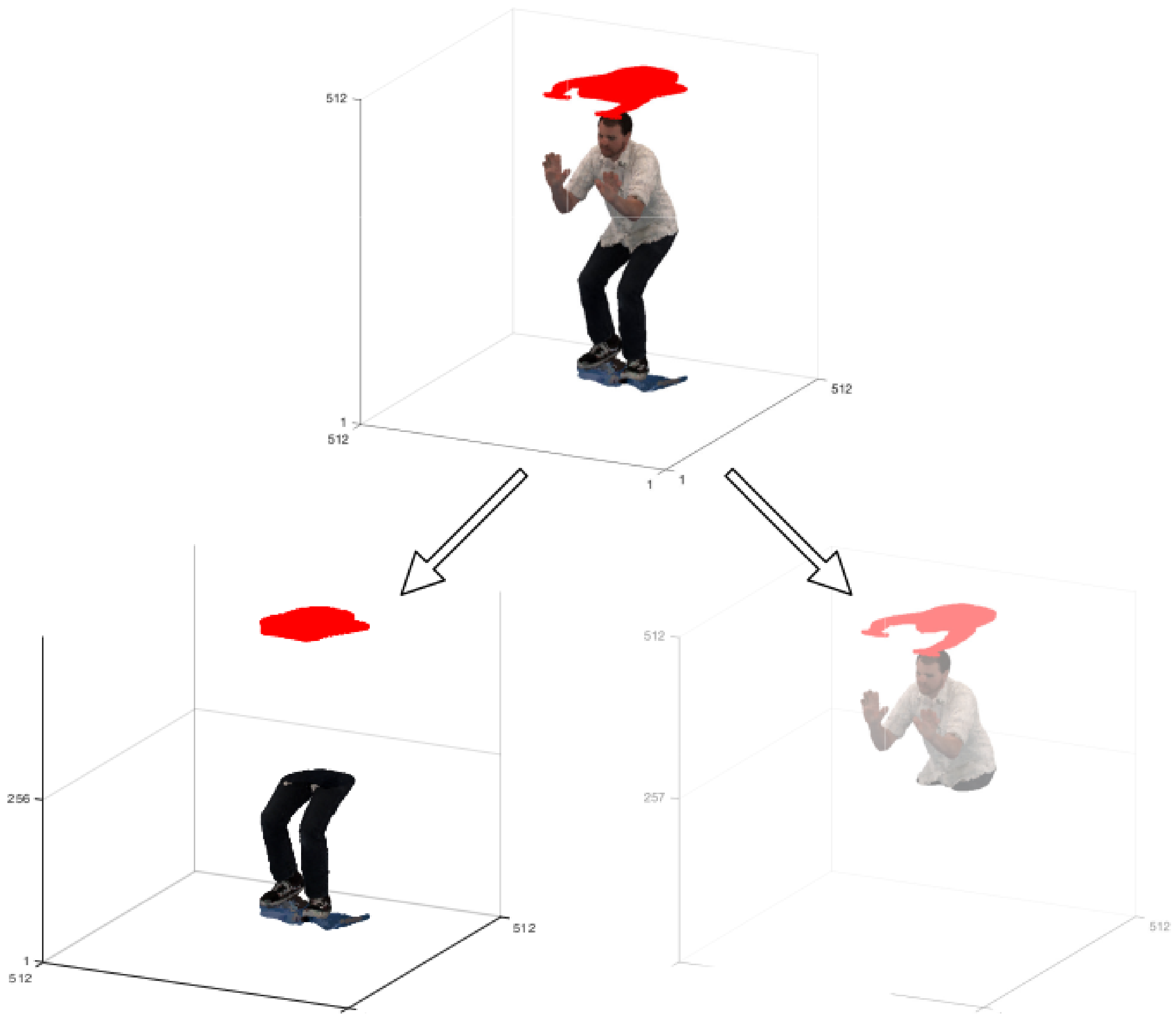


Figure 1.4: Dyadic decomposition on a single point cloud ilustrated.

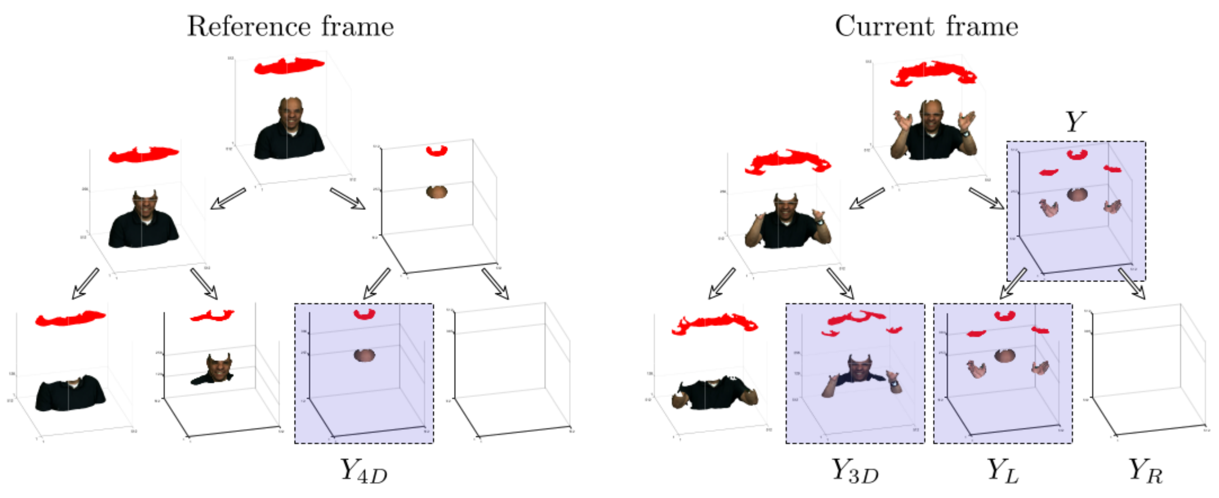


Figure 1.5: S4D decomposition process and context gathering.

bpov in point-to-point error (C2C) metrics [10].

Evolving the **S3D** to dynamic point clouds, where there's an additional time domain expressing motion frames on the point cloud videos, an inter-frame lossless geometry coder of dynamic voxelized point clouds was developed and published on the same year. This new method is based on **S3D** algorithm but also adding a fourth pixels context on the encoding process. Because of that this new technique is called **Silhouette-4d** (**S4D**). Thus, in addition to the pixels contexts from current silhouette, the brother silhouette and father, the pixels from another frame is used assisting on the encoding, as depicted in 1.5. And from its results, it presents best compression performance of lossless comparing with well known techniques on JPEG Pleno Database [11].

All these new method proposals and great performance results, based on Dyadic Decomposition approach and S3D basis technique, presented a great perspective and opportunities to develop further works and studies on this algorithm. By these studies, it was sought optimize this technique and explore in more deep details the algorithm, allowing refined versions with better results and performance. Both the S3D algorithm, developed by professor Eduardo Peixoto, and S4D algorithm, developed by professors Eduardo Peixoto and José Edil Guimarães, have been build on *Matlab* platform. Because of the facilities, tools — such as debugging, data states, and a local shell — and its Data Structures with its simple syntax on matrix and vectors manipulation, its platform served as a good environment to develop the initial versions of the algorithm, mainly based on these 2D (images and tables from S3D) and 3D (point clouds) matrix entities and its operations, such as boolean **OR** operations and iterating each element from the entity. Moreover, Matlab offers libraries and functions that are attractive to the signal processing community, providing pre-built functions such as filters, transforms, and matrix operations ending up being an well known language on engineering community [12] [13].

1.3 PROBLEM

On the other hand, the Matlab, being run by a interpreter, has a slow execution, a characteristic typical from interpreted languages. Besides, its garbage collector [14] and automatic allocation and memory management on the data structures and variables diminish the memory control by the user. As a consequence, the execution time on the program gets longer, and the storage usage during the execution is underutilized. Furthermore, as bigger point clouds are used on the algorithm, more time is spent and depending on how much increase on the point cloud size input is, it can be impracticable time. Because of these limits the algorithm analysis could not be done in a satisfactory depth and close to it real performance, exploring its scope limits capacity.

Based on this issue, a migration of this original Matlab code to a C++ was motivated, considering its qualities. The first quality is its own characteristically performance, thanks to its compiled language. This allows even a extensive program like the S3D algorithm be executed on a interesting fast time once translated to machine language, compared to a interpreted language. Besides that, C++ allows a direct user management of the storage, giving more control to arbitrary free and allocate memory when the algorithm requires, in order to explore the best capabilities of the storage. Without the use of a garbage collection and lesser language's own memory management reduces the time overhead too, considering that the garbage collection is an additional program that adds an overhead on the program execution time. It's important to mention the C++ syntax and structures inspired on Object Oriented Programming (OOP) facilitates on the code development and fit the abstract structures from the algorithm to a real implementation using classes and data structures provided by the C++ libraries. For example, High Efficiency Video Codec (HEVC) and Versatile Video Coding (VVC) codes are made in C++, reinfocing the importance of this language.

Hence, professor Eduardo Peixoto and José Edil Guimarães started to write a C++ code migration of the original Matlab S3D algorithm in 2020, and during the development, a team was selected in order to support and accomplish this migration considering the complexity of this task process and concerning a code with a satisfactory performance. At the beginning of the work of this new team, some classes and its function templates was already developed.

1.4 PURPOSE

The main goal of this project it to develop an initial and functional code, migrating the original Matlab S3D program and adapting its main data structures to the C++ environment and the language's data structures. This first version does not cover some features from the original algorithm, as the *single mode*.

As a member of this project, this work is focused on the project part responsible for design, modelling and implementing the API of the main data structures and classes required to the S3D algorithm, adapting the original code and adding new features to optimize the performance and usability on the S3D code development.

As experimental results measures that will allow to assess the C++ program migration and compare to the original Matlab code, the execution time and data compression rate will be compared in order to verify the validity and correctness of the code migration. It's expected that the time will be reduced and the compression rate doesn't be much different from the original. With this new migrated S3D implementation version, our purpose on this project is to submit the S3D algorithm to the MPEG's G-PCC for consideration, considering the out-performing result from the original Matlab implementation. Once the C++ version is a optimization over the original, a time performance improvement is expected.

CONCLUSION

We've seen the point cloud importance on the industry and different commercial applications. Moreover, how the coders to these new kind of data is a high demanding matter to its compression to a more efficient and practicable storing, processing and rendering. Aiming to offer a solution that gets better for each new coder proposal, different kind of strategies and approaches is being studied and developed in the academic and research field. One of these proposal is the S3D technique, whose results outperforms the all *state-of-art* intra coders tested, and is implemented on Matlab. This work aims to migrate this Matlab algorithm to a C++ implementation, a more usual language on the codec developing, with a more performative execution.

In order to migrate the Matlab original code, it's important not only to examine its project structure and how it's implemented, but also understand all the theory background behind the concepts involved on the S3D implementation. Thus, the **Literature Review** chapter purpose is to cover all the concepts required to understand the S3D algorithm. With a better comprehension on the algorithm, we are able to implement it by availing and pursuing all the performance and advantages that C++ can bring, which will be covered on **Implemented Solution** chapter. It's performance and execution will be assessed in **Results** chapter and then discussed and concluded in the **Conclusions** chapter. Some examples from the **Literature Review** can be verified on the **Appendix** chapter, bringing a better understanding on the concepts involved.

2 LITERATURE REVIEW

2.1 POINT CLOUDS

A point cloud can be defined as a set $\mathcal{V}$ of points [1]. A common process called *voxelization* delimits the point cloud's points in a 3D discrete grid, where each unit volumetric cube space from this grid is called *voxels*, which is used on the codec implemented on this work. The voxels are a 3D equivalent to image pixels, thus representing index triples corresponding to a 3D spatial location within a 3D grid dimension $2^k \times 2^k \times 2^k$, where k is the level within a voxel hierarchy $\mathcal{V}^k$. Each voxel can have multiple attributes associated, as color, normal, reflectance. The voxelized point clouds are captured by special cameras and processed by a special method called *voxelization*, assigning the attributes values and its geometry information. The number of voxels from a point cloud can be denoted as $|\mathcal{V}^k|$ notation. This measure is proportional to the object surface area. The maximum number of levels is called *depth*, which is determined by the number of bits used on the coordinate representations [15].

The point clouds can be classified as static or dynamic type, depending on their captured environment. The static type represents a static data point cloud time, i.e., doesn't express motion and time change. On the other hand, the dynamic type adds the time attribute domain, expressing motion and point cloud changing by its frames, crucial information to 3D videos, autonomous navigation based on 3D large scale dynamic maps and VR broadcasting [1]. By these types, MPEG PCC has separated the point cloud in three categories, in order to explore compression technologies according to the corresponding 3D applications, which each one has its own particular requirement, figure 2.1. The category 1 classifies static point clouds, the category 2 the dynamic point clouds, and finally the category 3 consists of dynamically acquired or fused point clouds [1].

Based on MPEG PCC standards, the point clouds files are represented by Polygon File Format (PLY) [1]. In this file type format, each voxel has its position (geometry information) and associated attributes. In general, these attributes describes the voxels properties such as colors and reflectance. We can observe that in the figure 2.1 [1], each category requires certain attributes. In order to visualize the point clouds pattern, an interesting free and open source software used to render point clouds, specially `.ply` files types, is the Meshlab. The PLY format 3D objects as a collection of vertices, edges and - depending on the volume representation - other elements. Each one of these collection may have properties as mentioned. Typically, a PLY file describing a point cloud contains a list (X, Y, Z) triples for vertices, along with its fixed properties specified on the header. The basic point cloud PLY file structure is similar to:

```
ply
format ascii 1.0
element vertex 213609          {we have a point cloud of 213609 voxels}
property float x
property float y
property float z
property uchar red
property uchar green
property uchar blue
end_header
223 255 215 96 63 46
223 255 216 108 75 58
223 255 217 128 83 60
```

This point cloud is a piece of the third frame from Ricardo9 file sample file from the JPEG Pleno Database [16]. This file begins with the header. The first line is the standard characters for the PLY format. The second line specifies the file format, an alternative format could be binary file. The next line describes the elements from the object. In this case, the element described is the vertex followed by its amount on the file, which is 213609. Then, a list of properties are described, indicating its type and the property name. Since the object described is a point cloud, no more elements are listed. Then the header is delimited by the `end_header` line. Thus, the following lines lists the elements described on the header following its declaration order, assigning the properties values corresponding for each element type. In this case, the following 213609 describes the point cloud's voxel. And for each one, its position determined by (X, Y, Z) coordinates and its color attributes (R, G, B) are described [17].

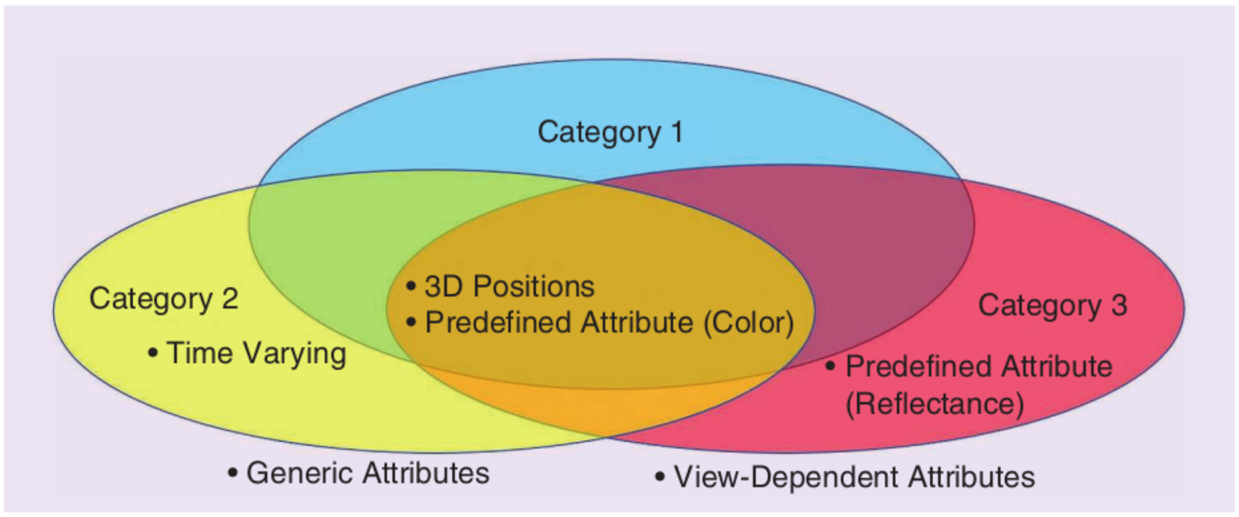

Category 1
Category 2
• Time Varying
• 3D Positions
• Predefined Attribute (Color)
Category 3
• Predefined Attribute (Reflectance)
• Generic Attributes
• View-Dependent Attributes

Figure 2.1: The three point cloud categories and its representation requirements. The not-yet-supported functionalities are outside of ellipse shapes.

2.2 SIGNAL COMPRESSION

This section will cover signal compression fundamentals and important concepts in order to understand the proposed point cloud compression algorithm steps and its performance results analysis in the next sections. Its content will be based on Sayood's book Introduction to Data Compression [18].

In the case of *compression algorithms*, we can consider that we are referring to two algorithms that composes it. There is the *compression algorithm* that takes as an input $\mathcal{X}$, carrying data, and generates a representation $\mathcal{X}_c$ representing the same data with fewer bits, i.e., a compressed data. Generally, we can refer to it as *Encoder*. And paired with this algorithm, we have a *reconstruction algorithm* that operates on the compressed data $\mathcal{X}_c$ to generate the reconstruction $\mathcal{Y}$. For this, we can refer as *Decoder*. Based on the requirements of the reconstruction and depending on its application, the data compression can be classified in two classes: *lossless* algorithms, where $\mathcal{Y}$ is identical to $\mathcal{X}$, and *lossy*, which $\mathcal{Y}$ is different from $\mathcal{X}$, but on the other hand generally provides higher compression than lossless. Considering the development of data compression algorithms for a variety of data, we can divide the process in two phases. The first one is the *modelling*, which we try to identify redundancies that exist on the data and describes these redundancies in the form of model. Following the modelling, we have to describe the model and how this encoded model differs from the original data, phase called *coding*. This model is encoded generally by a binary alphabet. Besides, the difference between the data and the model is often referred to as the *residual*.

The main measures that is focused on concerning to compression analysis is the amount of compression and how closely the reconstruction resembles the original data. The *compression rate* express the ratio between the size of the original data before compression to the size occupied by the new compressed data. We can also represent the compression ratio by expressing the reduction in amount of data required as a percentage of the size of the original data:

$$CR = \frac{S_o}{S_c} \tag{2.1}$$

or

$$CR = 1 - \frac{S_c}{S_o} \tag{2.2}$$

where CR is the compression rate, S_o is the original data size in bits and S_c it the compressed data size.

Another important metric is *rate*, which express the average number of bits required to represent a simple sample. In the case of point clouds context, a notable metric is the *bits per voxel occupied*, that is calculates the ratio between the number of bits occupied by the compressed file and the number of voxels occupied by the point cloud's file.

$$R = \frac{S_c}{N} \tag{2.3}$$

where R is rate, S_c it the compressed data size and N is the amount of voxels occupied.

In lossy compression context, the reconstruction differs from the original data. Therefore, in order to determine the efficiency of the compression algorithm in reconstruct the data qualifying the difference between the original data and the reconstructed data, we measure the *distortion* between the data.

In order to analyse and have a further study on the point cloud compression method, it's important to understand the concepts and principles from the information theory. We can recognize the information theory as a mathematical basis for compression methods. An important concept from this study area is the quantity called *self-information*. Suppose we have an event A, which is a set of samples of some experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by:

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A) \tag{2.4}$$

The base b determines the unit of the self-information. If its value is 2, the unit is *bits*; e, for *nats* and 10 for the unit *hartleys*. The idea behind this logarithm function is to express that when a certain event has a high probability to happen, it adds little information from what we already know and expect from a case. But if the event has low probability, it adds a lot of information from the analysis, i.e., it contains high self-information associated. This is because when a certain event that we don't have a lot of expectation to occur happens, it indicates in the sampling analysis that a new information is added to the amount of information that shares a common idea or a specific tendency around a value or event. And this new information is an occurrence that differs from the past common events.

Given a set of independent events A_i, which are a set of outcomes of some experiment $\mathcal{S}$, such as:

$$\bigcup A_i = S \tag{2.5}$$

where S is the sample space, then the average self-information associated with the random experiment is given by:

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i) \tag{2.6}$$

This average is called the *entropy* associated with the experiment. Shannon noted that if we have an experiment as a source that put out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average number of binary symbols needed to code the output of the source. Moreover, the entropy of a source data express the best number of bits average of an encoded output that a lossless compression algorithm can do on a given source.

Having a good modelling for the data can be useful in estimating the entropy of the source. Besides, good models leads to more efficient compression algorithms. Thus, in order to develop techniques that manipulates that by mathematical operations, a mathematical model for the data is crucial. Considering this, the better the model matches with the aspects of reality of the data represented, the more likely the technique will present a satisfactory technique. The main approaches possible are a *Physical Model*, *Probability Model* and *Markov models*.

The characters representation of a given source data is referred to a symbol. And from a given set of symbols $\mathcal{A}$, its called *alphabet*, and the symbols as *letters*. Using this alphabet, the coding phase performs an assignment of binary sequences, called *codewords* to the letters from the alphabet. This set of binary sequences is called *code*.

2.2.1 Geometry compression

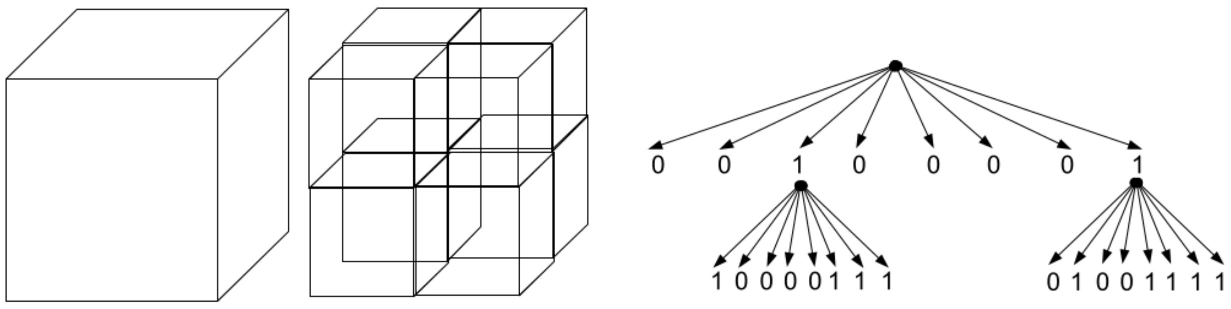
Given a brief overview on the main signal compression area concepts, its adequate with all this basis apply this comprehension on the problem that will be studied. It's been already explained that with a compression algorithm a data can be compressed to a file with lesser bits, and reconstructed from this compressed file; and this algorithm can be divided in two main phases: the modelling and coding process. By the model adopted, the coding process converts the data source's letters (from a given alphabet) to a associated sequence of binary sequences called codewords (from a given code). With the algorithm modelled and implemented, in order to examine its performance, a set of important measures can be used to its analysis and benchmark with other algorithms. One of them is entropy, in the case of lossy compression algorithms, distortion. But in the point clouds case, based on all this signal compression theory, which data is being compressed, considering that it may carry a diversity of information? Depending on which category the point cloud are being used as input to the algorithm, different attributes is crucial to be encoded on the compression. We could mention the color, geometry, reflectance. In the case of the algorithm that will be addressed, only the geometry will be the object of compression. Thereafter, its convenient to comprehend what a *geometry compression* is about.

When a point cloud is rendered, its volume shape and its surface contour can be visualised and described. This description can explore some similarities between certain sections of the volume, the amount of space occupied, and the empty space. Furthermore, these set of observations and its exam allows the discovering of side information and, from them, ways that they can implicitly express by the context of the volume being analysed some other information from the volume. Besides, there's the possibility of the existence of some information that can be removed from the 3D model without compromising the original content. It's a well known technique used beyond the point cloud compression as in image and video encoding.

Thereby, the geometry compression focus on comprising the 3D shape and contour occupied, in the case of the work, by the point cloud. And this compression explores the information and redundancies present on the 3D object associated with the algorithm approach adopted. It's important to note that the geometry compression not restricts only to point clouds, but to other kinds of 3D objects representations as meshes. In this case, the algorithm can explore more information concerning to the volume, as the connectivity between the vertices. An important and notable geometry compression algorithm example to mention is the octree approach [7], where recursively the point cloud is divided in 8 cubes, until it reaches the voxel level. The structure of the decomposition is a extension of 3d quad-tree [4]. For each volume occupied, its marked as 1 in the tree, and where is empty, as 0. These empty volumes are treated as leafs from the tree, figures 2.2a and 2.2b [4].

2.3 CABAC

Once a geometry compression is made on a given volumetric object model, a significant amount of bits is saved to represent it's data. Nonetheless, only the geometric compression usually is not sufficient in order to make a efficient compression. Given the bits that represent the point cloud geometry, it's possible



(a) Unit cube divided into 8 sub-cube.

(b) The volume encoding process.

Figure 2.2: The octree geometry compression approach illustration [4]

to explore classic signal compression algorithms that can be performed on the geometry compressed bits. From these classic algorithms, two main types of code can be generated from them. Those are: *fixed-length* code and *variable-length* code. The fixed length code express all the codes where all symbol from the alphabet is represent by codewords with a same amount of bits. On the other hand, the variable length codes represent the codes where each symbol has it's own amount of bit, some of them is represented with less bits, reducing the overall compressed size.

An important classic variable length compression algorithm used on the proposed point cloud compression is the *Context Adaptive Binary Arithmetic Coder*. This algorithm will be covered and clarified on this section. It will start from the basic implementation, and then scaled to the algorithm that is aimed.

2.3.1 Arithmetic Coding

The *arithmetic coding* is a variable length coding useful when dealing with small alphabets, such as alphabets with highly skewed probabilities and binary sources, as the case of the geometry compressed bitstream on this work. Moreover, it's useful approach when it's required to separate the modelling and coding aspects from a lossless compression.

The idea behind the arithmetic coding is an important property that allows a more efficient coding. When the codewords is generated for groups and sequences of symbols rather than to each separated symbol in a sequence, it will result in a more efficient compression on the final encoded sequence. Futhermore, concerning a more viable way to assign codewords to particular sequences without having to generate codes for all sequences, a particular approach that would result in a huge amount of side data; the arithmetic coding proposes an unique identifier. This identifier is called *tag*. This tag is a unique value that will be used to help the encoding and decoding process. Thus, first the tag is generated from a given sequence of symbols, then it's assigned an unique binary code. From this unique binary code, it can be deciphered, and recover the original sequence.

With this overview, it will be explained the **coding process**. In order to distinguish the sequence of symbols, we need a tag with an unique identifier. A possible set of values to be assigned to a tag is the numbers in the interval $[0, 1)$. Since it contains an infinite amount of different numbers, it allows to assign an unique tag, in this case a decimal value, for each different sequence of symbols. But for this process, it's firstly required a function that maps a sequence symbols to a value from this unit interval. In the case of the arithmetic coding, the cumulative distribution function (*cdf*) comply with this requirement, establishing the association between the random variable representing the symbol and its probability.

Hence, given an alphabet $\mathcal{A} = a_1, a_2, ..., a_m$ for a discrete source and X a random variable:

$$X(a_i) = i,\ a_i \in \mathcal{A} \tag{2.7}$$

we have a probability model $\mathcal{P}$ that express the probability density function for the random variable:

$$P(X = i) = P(a_i) \tag{2.8}$$

and the cumulative density function defined as:

$$F_X(i) = \sum_{k=1}^{i} P(X = k) \tag{2.9}$$

With this given notation, we can describe the **tag generating process**, which corresponds to the encoding process to the arithmetic coding. The tag generating process works by reducing the interval size in which the tag is being generated as more the elements of the sequences is received. First, we divide the unit interval into sub-intervals of the form $[F_X(i-1), F_X(i)), i = 1, ..., m$. We can remember that the unit interval ranges $[0, 1)$, thus this partition divides exactly between this interval. For each symbol a_i the sub-interval $[F_X(i-1), F_X(i))$ is associated. Then the source elements is consumed. For each element received, for example a_k, the corresponding interval $[F_X(k-1), F_X(k))$ is divided in the same proportions as the original interval, and the tag is restricted on this new interval. This process repeats as each symbols from the source is received.

For example, consider a three letter alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with $P(a_1) = 0.5$, $P(a_2) = 0.2$ and $P(a_3) = 0.3$. Thus $F_X(1) = 0.5$, $F_X(2) = 0.7$ and $F_X(3) = 1$. And consider a sequence $\{a_1, a_2, a_3\}$. Following the algorithm steps described above, the intervals obtained would be Figure 2.3. It's important to note that the intervals generated are disjoints from all possible intervals obtained when encoding other sequences.

We can denote the source sequence with length n as $(x_1 x_2 ... x_n)$. And the sequence has been codified as $\mathbf{x} = (x_1 x_2 ... x_k)$, where the coding has been made until the *kth* element. Thus, the lower bound of the interval where the tag resides l at *kth* element is denoted by:

$$l^{(k)} = l^{(k-1)} + (u^{(k-1)} - l^{(k-1)}) F_X(x_k - 1) \tag{2.10}$$

and similarly, the upper bound u at *kth* symbol :

$$u^{(k)} = l^{(k-1)} + (u^{(k-1)} - l^{(k-1)}) F_X(x_k) \tag{2.11}$$

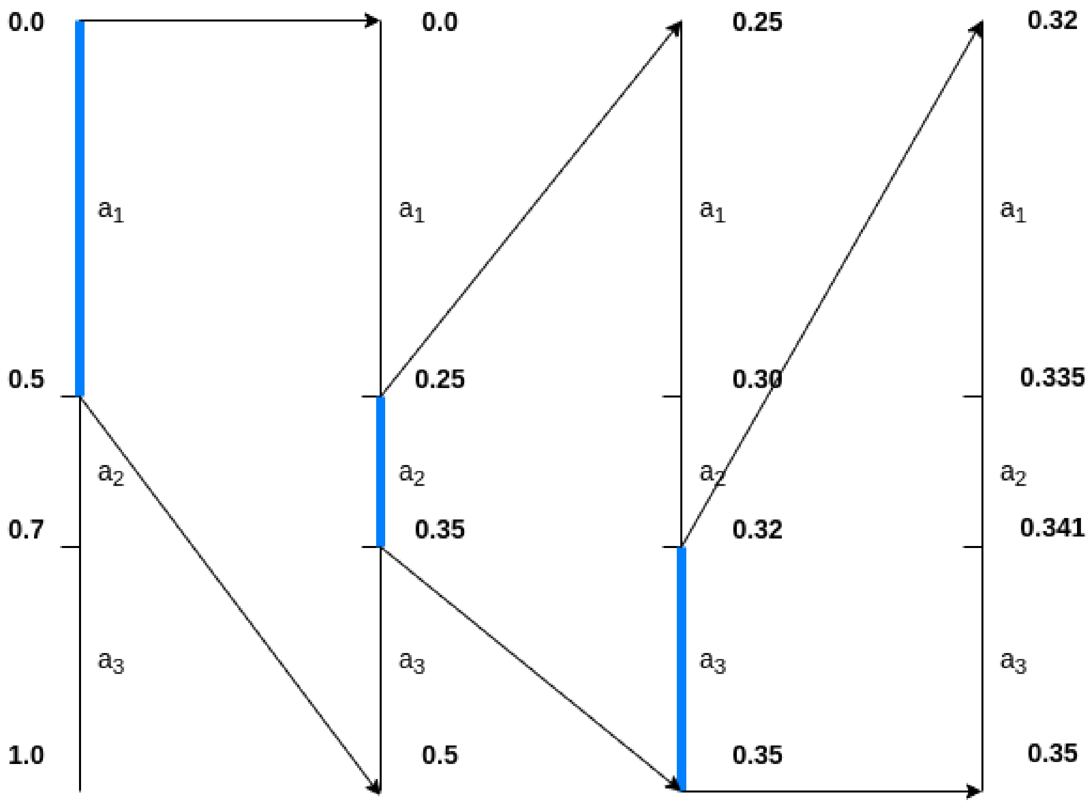


Figure 2.3: Arithmetic coding tag's interval generating for sequence $\{a_1, a_2, a_3\}$

Being the tag restricted on the final interval determined by the last symbol, we can denote the tag value as the midpoint of the interval for the tag. Then, the tag value $\overline{T}_X(\mathbf{x})$ is:

$$\overline{T}_X(\mathbf{x}) = \frac{u^{(n)} + l^{(n)}}{2}$$

We can overview the encoding as:

1. Initialize $l^{(0)} = 0$ and $u^{(0)} = 1$.
2. For each k consumes source elements and obtain $l^{(k)}$ and $u^{(k)}$
3. update $l^{(k)}$ and $u^{(k)}$
4. Continue until the entire source n length sequence has been encoded
5. Computes tag value as $\overline{T}_X = (u^{(n)} + l^{(n)})/2$

In order to generate the encoded source data, the tag value is truncated and converted to an fixed point binary representation. Then, this is the encoded data result.

Given the tag generating process, equivalent to the encoding of arithmetic coding, we can now recover the original data sequence with this given tag. Thus, the **tag deciphering** will be described, which is equivalent to the decoding. The decoding is a reproduce of the encoding steps. It's important to note that the interval containing the tag value is a sub-interval of every intervals obtained in the encoding. That being so, the strategy behind the decoding is to decode the source elements such way that its consequent interval, $u^{(k)}$ and $l^{(k)}$, contains the tag value for each *kth* element, x_k.

We can overview the decoding as:

1. Initialize $l^{(0)} = 0$ and $u^{(0)} = 1$.
2. For each k find $t^* = (tag - l^{(k-1)})/u^{(k-1)} - l^{(k-1)}$.
3. Find the value of x_k for which $F_X(x_k - 1) \leq t^* < F_X(x_k)$.
4. updade $u^{(k)}$ and $l^{(k)}$.
5. Continue until the entire sequence has been decoded.

It has been shown in Sayood[18] that:

$$l_A < H(X) + \frac{2}{m} \tag{2.12}$$

Having a rate close to entropy as the source sequence length increases.

The shown implementation, however, is very limited. Considering the limit of values allowed to a computer in the range $[0, 1)$, the unlimited set of unique values that a tag can assume when encoded as fixed point representation does not remain valid when its value is not supported by the computer decimal number limit precision. Besides, the bounds, limited by these restrictions, is prone to converge by the truncation. Under those circumstances, an possible strategy is rescale the intervals when its needed. This approach satisfies an synchronized rescaling - rescale preserving the information being transmitted - and incremental encoding - transmit portions of the code as its read, instead of wait until the entire sequence is read.

This approach has three possibilities as the interval gets narrower:

1.
2. The interval is entirely confined to the lower half of the unit interval [0, 0.5).
3. The interval is entirely confined to the upper half of the unit interval [0.5, 1.0).
4. The interval straddles the midpoint of the unit interval.

The idea is remap the sub-interval to an new interval with range $[0, 1)$. For each case we do the following:

1. Send bit 1 to decoder, and remap $E_1 \ : [0, 0.5) \longrightarrow [0, 1);\ E_1(x) = 2x$
2. Send bit 0 to decoder, and remap $E_2 \ : [0.5, 1) \longrightarrow [0, 1);\ E_2(x) = 2(x - 0.5)$
3. Just proceed

2.3.2 Context Arithmetic Coding

A possible improvement can be added on the Arithmetic coding. Let's consider for example a sequence:

$$(a_1, a_2, a_2, a_2, a_1, a_3, a_3, a_1.a_4, a_1, a_3, a_1)$$

If we analyse a_2 frequency on this sequence, we can observe it's presence only 25% of the total occurrence. But if we already know the previous symbol, it gives more precision on the probability of its occurrence. In this sequence, the a_2 occurs on two cases: preceding by a_1 or a_2. Moreover, the first case occurs 33% of the occurence, while the second 66%. Thus, if it's known in advance that the preceding symbol is an another a_2, the probability of a_2 increases from 25% to 66%. It's adding information on the probability estimate is called *context*. Thus, with this adding information, more data can be saved considering the data that has already been encoded/decoded and it's probability relation, giving a more power of self-deduction by the algorithm to predict the next symbol based on the previous symbols. An example is well explained in Appendix section 6.2.

2.3.3 Context Adaptive Arithmetic Coding

In the above example, we've assumed that the context probability table is already available and computed. But in the most cases, that's an information that actually is not available to the encoder. Thereby, the algorithm must be adapted to new learnt distribution as the coding progresses. A simple strategy is initialize the table with all elements as counters initialized with 1. So at the beginning, few information about the source is known. For each symbol encoded, the counter is associated with the context is incremented. The same mechanism is reproduced on decoder. After each symbol encoded, the counter is updated accordingly.

For example, let's consider a source with an alphabet $\mathcal{A} = a_1, a_2, a_3$ and we need to decode the message:

$$(a_1, a_1, a_2, a_1, a_3)$$

The context table is initialized as, where each line is a context:

Context/Symbol	a_1	a_2	a_3	**Total**
None	1	1	1	3
a_1	1	1	1	3
a_2	1	1	1	3
a_3	1	1	1	3

After encoding the first symbol a_1, the table doesn't change since it's the first symbol with no contexts. Then after the second symbol a_1 we have 2.1.

The process repeats until all symbols are decode, having the table at the end 2.2

Table 2.1: Table after encoding a_1

Context/Symbol	a_1	a_2	a_3	Total
None	1	1	1	3
a_1	2	1	1	4
a_2	1	1	1	3
a_3	1	1	1	3

Table 2.2: Table after encoding all symbols

Context/Symbol	a_1	a_2	a_3	Total
None	1	1	1	3
a_1	2	2	2	6
a_2	2	1	1	4
a_3	1	1	1	3

At the end all the contexts are updated, with its probabilities adapting for each symbol encoded.

2.3.4 Context Adaptive Binary Arithmetic Coding

With all the arithmetic encoding covered until now, the theoretical basis is sufficient to introduce the main encoder that is crucial to the encoding of the binary point cloud geometry encoded sequence on the S3D approach. We now will cover the **Context Adaptive Binary Arithmetic Coder**(**CABAC**).

Not just to the point cloud case, but in many applications the alphabet is itself *binary*, such as bilevel documents, or binary representation of nonbinary data, as in the case of H.264 CABAC. And the main advantage of having a probability model with only two letters in the alphabet is that the probability model consists of a single number, consisting the probability of just one of the symbols. Having this value, the other symbol probability is just 1 minus this probability. As a consequence of this, the requirement of less values to represent the probability model allows the use of multiple contexts to encode the source sequence, which gives a better compression method.

This arithmetic coder combines the *Context Adaptive* approach mentioned and the *binary arithmetic coder*. The Context Adaptive part of the algorithm follows the same mechanism. Thus, the first thing in this algorithm we have to decide is the word length to be used. Given a word of length of *m*, the values of interval $[0, 1)$ is mapped to the range of 2^m binary words. The 0 is equivalent to 00...00 *m* times, and 1 to 11..11 *m* times, and the value 0.5 as 100..000, with *m-1* zeros. For example, if we have a word of 8 bits, 0000000 corresponds to 0, and in the interval 0.0. And for 10000000, 128, and in the interval 0.5. And for 11111111, 255, corresponding to 1 in the interval. Thus, the bound will assume these mapping values during the algorithm.

Since we do not know previously the message length - the total count - most of the cases, and using an Adaptive Case approach; we have to pick the word length *m* independent of the message length. Also, it's demonstrated on Sayood that given a word of length *m* we can only accommodate a total count of 2^{m-2} or less. Thus, when the total count of symbols encoded approaches 2^{m-2}, we have to rescale the interval.

This can be performed dividing all numbers by 2 and rounding up the results so that no value is rescaled to zero. Besides, this periodic rescaling can have benefits on the count table by refreshing and reflecting better the local statitic of the source.

The interval computation, by consequence of having only 2 value, can be simplified by updating only one endpoint and the interval's size. Thus, storing the lower end of the interval l^n and the size of the interval A_n, where:

$$A^{(n)} = u^{(n)} - l^{(n)} \tag{2.13}$$

the tag for a sequence is the binary representation of $l^{(n)}$. And instead of treating the values as 0s and 1s, it's common to deal with these values as More Probable Symbol (MPS) and Least Probable Symbol (LPS). If we denote the probability of occurrence of the LPS for the context C by q_c and mapping the MPS to the lower subinterval, the occurrence of a MPS symbol results in the update equations:

$$l^{(n)} = l^{(n-1)} \tag{2.14}$$
$$A^{(n)} = A^{(n-1)}(1 - q_c) \tag{2.15}$$

and for the occurrence of LPS symbol, the update is described as:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c) \tag{2.16}$$
$$A^{(n)} = A^{(n-1)}q_c \tag{2.17}$$

Considering the mapping of the $[0, 1)$ interval values to binary words, an important property to note is that if the two bounds, $l^{(0)}$ and $u^{(0)}$, are in either on the upper half or lower half, they shares the same most significant bit (MSB). Where, if the bit is 1, it's in the upper half; and the bit is 0, lower half. With knowledge of this property, the mapping E_1, E_2 and E_3 becomes a bit shift operation. So, encoding steps becomes:

1. If MSB from both bounds is equal, shifts each bound 1 bit to the left.

   If it's 0, we will perform E_1

   If it's 1, we will perform E_2

2. If the 2nd MSB from $u^{(n)}$ is 0 and $l^{(n)}$ is 1, we will perform E_3. Shifts each bound 1 bit to the left, then complement the MSB.

   For every case, add bit 0 at right at $l^{(n)}$ and bit 1 at $u^{(n)}$.

The bounds updating equations becomes:

$$l^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_count(x_n - 1)}{Total\_count} \rfloor \tag{2.18}$$

$$u^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_count(x_n)}{Total\_count} \rfloor - 1 \tag{2.19}$$

With $Total\_count$ being the amount of symbols read on a given context, and Cum_count the amount of times that a specific symbol occurred on the last read symbols on a given context.

The decoding follows the same logic from previous coders, except that the t^* value is given by:

$$t^* = \frac{(t - l + 1) \times Total\_Count - 1}{u - l + 1}$$

Where t is the tag value. To a more clear explaining of each step of the algorithm and its mechanism cases, an example is illustrate in Appendix section 6.3.

2.4 SILHOUETTE 3D

Once we had an overview on the main topics and concepts involving signal compression and understanding the *CABAC*, an important algorithm that will be used on this work; we will finally cover the main algorithm responsible to the point cloud compression. But, before we study in more details the geometry compression part of the algorithm, it's important to review some other Point cloud geometry compressors.

One of the main compressors for point cloud is the octree based, which consists in recursively cut the point cloud in little 8 cubes composing the entire volume [7]. The MPEG G-PCC geometry compression is based on the octree proposal, using a arithmetic coder over the geometry compression. Others codecs variations explore the octree approach using different kinds of arithemtic encoders[4]. Another approach we could cite is using volumetric functions, based on B-spline wavelet [19] to code those volumetric functions representing both geometry and attributes. Also, some approaches uses the deep learning, processing over the raw data into codewords [6]. We can observe from these approaches that most of them are based on explore volumetric attributes and properties on the point cloud representation that can be removed considering its redundancies or relations between parts from the volume. Or on the case of the deep learning approach, we can abstract the point cloud as a sequence of symbols that can be encoded as any other symbols sequence, but also availing the point cloud symbols sequence characteristics on compression. The S3D proposal has its particularity. Different from the previous approaches that explores the volumetric properties and pattern over raw data describing point clouds, the S3D uses the silhouettes from point cloud slices as a source of coding. Using those silhouettes as source of the geometry coding, and considering that those silhouettes generated along the process is binary images, it's very appropriate to apply binary images compression based techniques on this approach. That's a notable characteristic that is applicable on the compressing process and gives efficient compression rate results [9], outperforming the G-PCC performance. We will explore the theory behind this technique on this section.

2.4.1 Silhouette

The main idea behind the *dyadic decomposition* approach is explore the geometry of the point cloud as a 3D occupancy array $N \times N \times N$, where the coordinates occupied coordinates is 1, and 0 otherwise. For this reason, it's appropriate to treat it as an 3D boolean array $G(x, y, z)$.

If we start so slice the point cloud along a given axis, and for each slice project on a $N \times N$ image all the points occupied on this region, we obtain an image similar to a *sillhouette* of this region from a point cloud. Thus, we can define a sillhouete as:

$$\begin{aligned} I(i,j) &= silhouette(G, axis, iStart, iEnd) \\ &= \begin{cases} \sum_{n=iStart}^{iEnd} G(axis, start, end) \text{ if } axis = x \\ \\ \sum_{n=iStart}^{iEnd} G(start, axis, end) \text{ if } axis = y \\ \\ \sum_{n=iStart}^{iEnd} G(start, end, axis) \text{ if } axis = z \end{cases} \end{aligned}$$

Where G is the point cloud represented by an 3D boolean array, the summation $\sum$ is done by through **OR** operation. By this operation, all slices in the interval $[iStart, iEnd]$ from the point cloud is merged into a single bitmap image $I(i, j)$. We can observe this ilustrated at Fig. 2.4

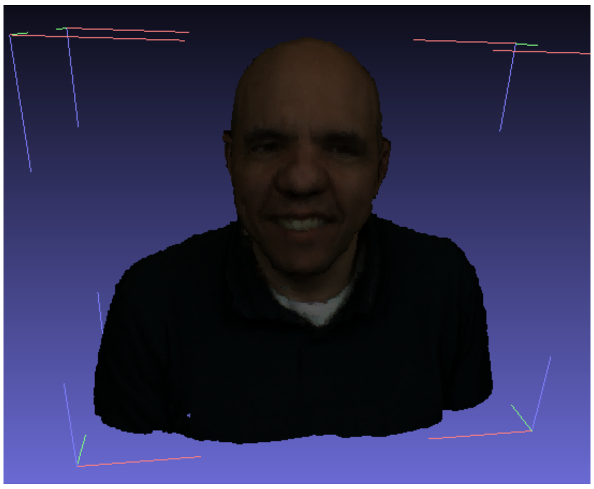
2.4.2 Silhouette tree

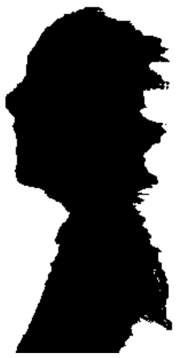
Having this silhouettes as elements describing the point cloud geometry from its slices, the algorithm works upon a recursive basis. In the beginning, the point cloud is sliced in two smaller intervals. And each slice is also divided in two intervals. On the case of this algorithm, the range of this intervals are the halves from the original slice. Thus, the point cloud is recursively divided in halves. This process is repeated until the slice does not contain any points inside, or the slice atomically has a width 1, being so a bitmap image.

Picturing this process, we get a binary tree. In this binary tree, each node is a slice from the point cloud, from which we can build a $N \times N$ silhouette projected by this slice, as we can see on Fig. 2.5a [10]. In this figure, each node represents a slice and the red shade a silhouette projection from the slice along the vertical axis.

2.4.3 Silhouette decomposition

Given the binary tree constituted by silhouettes describing the point cloud on each interval, the geometry compression based on *silhouette decomposition* initiates encoding each node's bitmap image from the tree in order to transmit the images that constitutes the entire point cloud. This decomposition takes advantage of one property from the silhouette generating process, concerning about the unoccupied voxels. If there's a blank pixel on a specific coordinate from the silhouette, that's because along the interval projection, none of the slices that composes the silhouette has that pixel occupied. We can deduce this from the OR summation process property, where if at least 1 slice has the pixel occupied, the projection will have


(a) ricardo9 point cloud rendered


(b) ricardo9 sillhouette along axis X

(c) ricardo9 sillhouette along axis Y

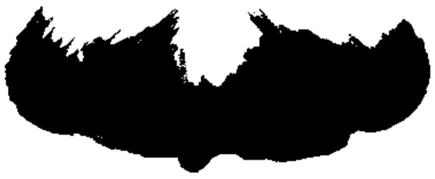
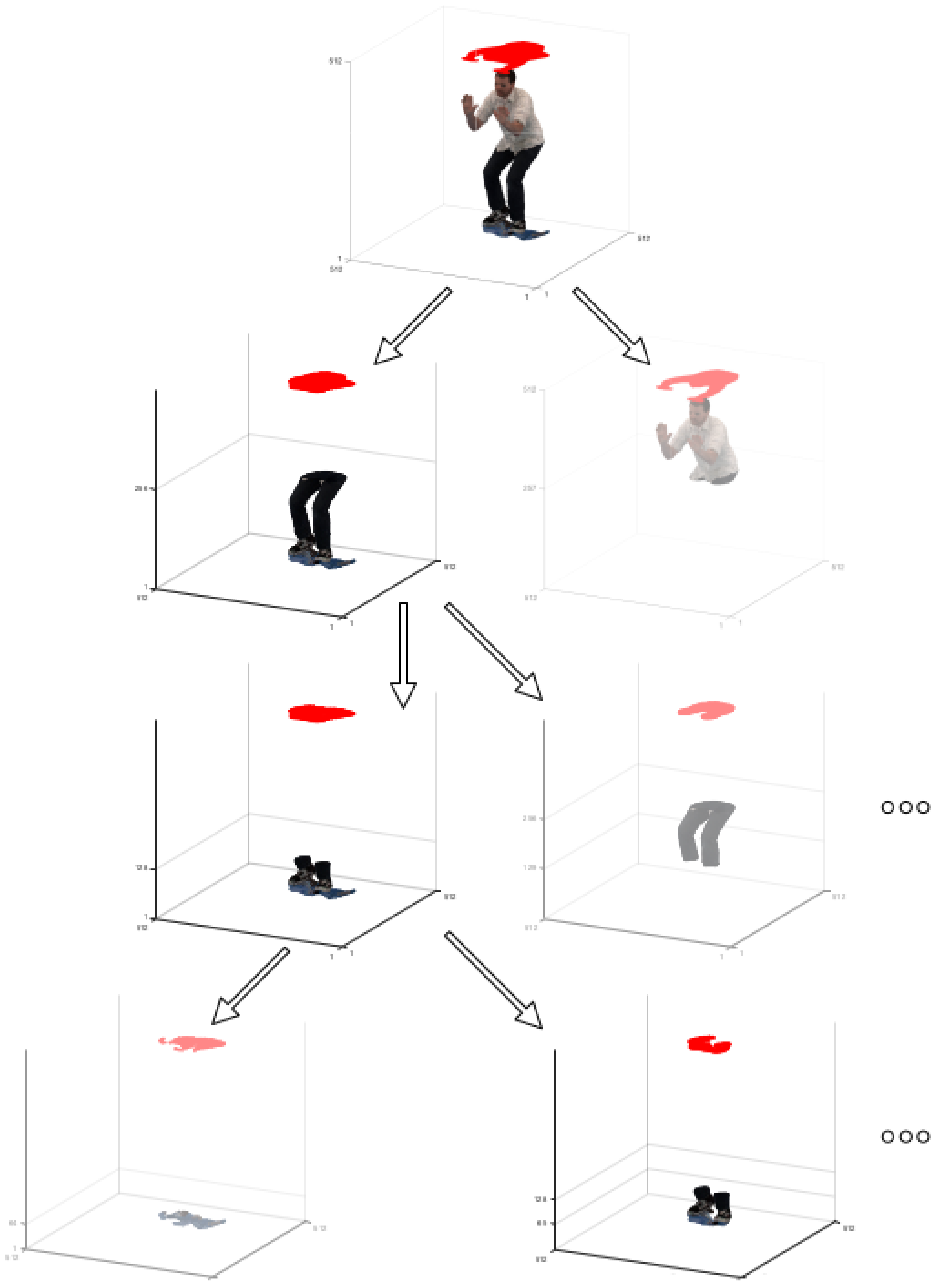

(d) ricardo9 sillhouette along axis Z

Figure 2.4: Ricardo9 point cloud and its silhouettes projections along entire axis


(a) Binary tree derived from the point cloud recursive slices.

this pixel occupied as well.

In the context of the tree, all the subtrees generated from a node silhouette will not have pixels occupied where this parent node hasn't. Besides, this decomposition takes advantage of the similarities shared between the neighbours node's silhouettes. These neighbours are the parent node and the other half generated from the slicing from the same parent, a "brother node". That's why the technique is called *dyadic decomposition*. It's important to note that this decomposition does not occur on the root node, since he has not parent node neither a brother node.

Given a node from the tree Y_C, and its children nodes Y_L (left child) and Y_R (right child) we will make the transmission of both children, and consider that Y_C was already transmitted. Then we do the following steps to transmit both images:

1. In the Y_L transmission, given the parent image Y_C as the mask of the decomposition, only the bits where the Y_C's value is 1 we send the from Y_L.
2. In the Y_R transmission, given the parent image Y_C and Y_L as the masks of the decomposition, only the bits where the Y_C's and Y_L's values is 1 we send the from Y_L.

The reason of we can do the 1st decomposition is remember that $Y_C = Y_L + Y_R$, where the sum is an **OR** operation. The only possibility where Y_C is 0 is when both values Y_L and Y_R is 0. Considering that Y_C was already transmitted, we can deduce these values to both children silhouettes. Thus we only send the bits where Y_C is 1. Furthermore, the explanation of the second decomposition is - considering that as Y_C, Y_L was already sent - we know that where $Y_C = Y_L + Y_R \Rightarrow 1 = 0 + Y_R = Y_R$, i.e, the bits where the parent is 1 and the left child is 0, the right child's bit should be 1. Thus we send only the bits where the parent and the right child is 1.

For a more clear understanding, consider the example in the figure 2.6

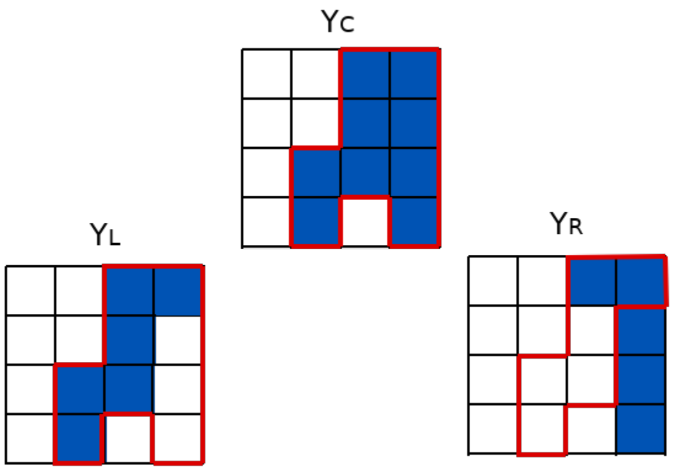


Figure 2.6: A simple silhouette decomposition ilustration

Having the blue pixels as the occupied pixels, the white pixels as empty and the red contour the bits where should be transmitted, we observe that the contour outlines only the bits where the mask delimit,

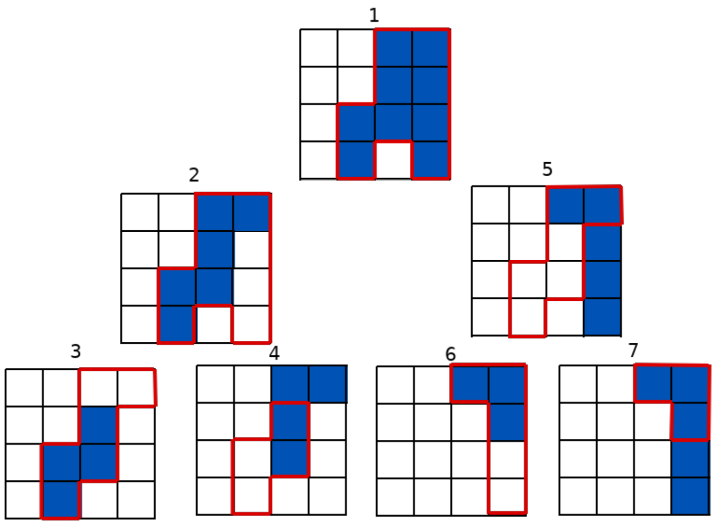


Figure 2.7: A silhouette decomposition example with 3 levels

which is the occupied pixels on the parent or left node silhouette. The entire decomposition from this example would be figure 2.7, where silhouette 1 is the root from the binary tree. Knowing that only the root silhouette is sent entirely, and the remnant silhouettes from the tree is sent by the decomposition, the bitstream sent from each node is:

1. 1: 0011001101110101
2. 2: 111011010
3. 3: 001111
4. 4: 1010
5. 5: 110000
6. 6: 11100
7. 7: 111

The final bitstream becomes 0011001101110101111011010001111101011000011100111, following the number ordering. This sequence occupies 49 bits, compared to send all the four 4×4 images, occupying 64 bits. It's important to note that the silhouette decomposition process mentioned follows a *preorder tree traversal*, decomposing and encoding from the parent node, to left child node and then right child node, recursively. Thus, the images from the point cloud are sent following this order.

2.4.4 Encoding

As the geometry encoding is done for each node from the binary tree, the idea to optimize the compression rate is perform a second level of encoding upon the bitstream generated from the previous step, in this case, using the CABAC.

The CABAC has 16 bits precision, and all its contexts are initialized only once with value 1. When the first image is transmitted and compressed with CABAC, the contexts used are the 10 pixels from the image itself as in Fig. 2.8 (a). These pixels contexts are called **2D contexts**. The image Y_L, following the tree notation, is encoded with the 2D contexts - 5 pixels - plus 9 pixels contexts from an additional image: the Y_C. Those are referred as **3D contexts**, as in Fig. 2.8 (b) [9]. And finally the image Y_R is encoded using the 2D contexts, with the additional 3D contexts provided by the Y_L pixels. These contexts is ilustrated at Fig. 2.9. This process is repeated until all nodes are covered, in preoder traversal, resulting in the final encoded bitstream.

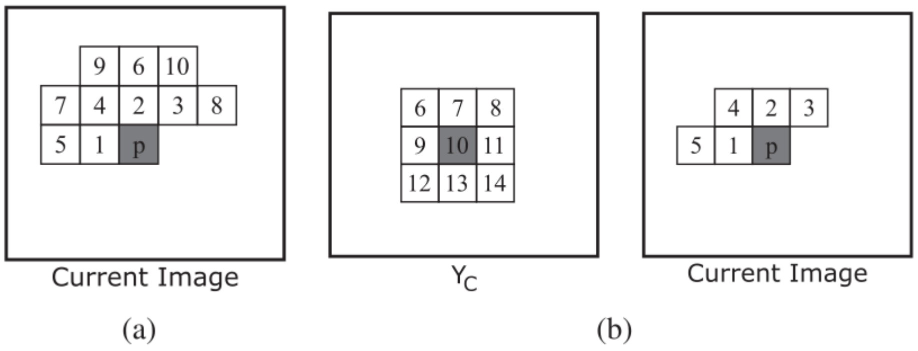


Figure 2.8: Contexts used to encode pixel p: (a) 2D Contexts and (b) 3D Contexts

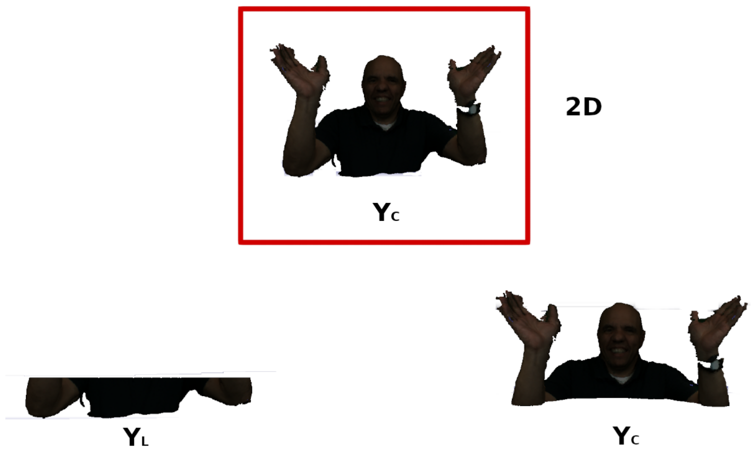
2.4.5 Decoding

The decoding process is the inverse to the encoding steps. First, the bitstream are decoded using the CABAC. With the knownledge of the tree traversal, it's possible to the decoder deduce which image from the tree is being decoded, and so use the correct 3D contexts from outside image's pixels. Recovered the geometry bitstream, it's possible to deduce the silhouettes from the point cloud that fill it's occupancy voxels, and once the leaf nodes are the slices - with width 1 - that when lined up and merged constitutes the entire point cloud, as depicted in fig.2.10, we can already recover the original point cloud without loss of voxels.

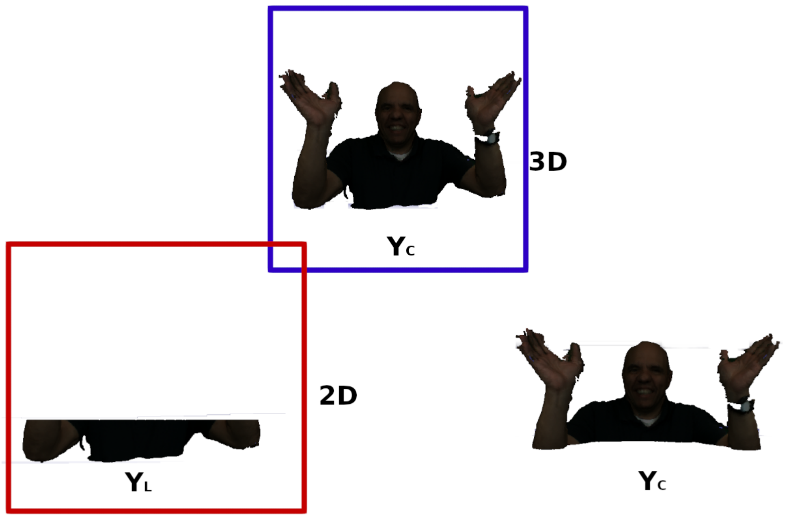
Conclusion

With this theory background covered, a deeper understanding on the S3D algorithm can be examined. Furthermore, a better notion of the algorithm mechanism and its concepts allows a more comprehensive analysis on its Matlab implementation, matching the abstractions, functions and modules derived from the code developing process with these concepts involved on the algorithm. Therefore, with all this basis, we are able to migrate its main ideas and design to a C++ perspective, availing its features and making some modifications in order to have a better performance, more structured project and befitting implementation with the original ideas proposed by the algorithm.

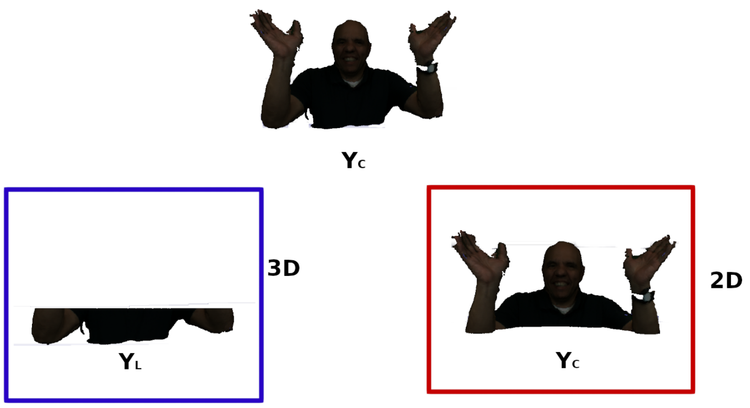
In the next chapter the C++ project will be described, covering an overview on its structure and focusing on the data abstractions and classes API design - this work main proposal. In addition, an explanation on



(a) Y_C encoding



(b) Y_L encoding



(c) Y_R encoding

Figure 2.9: The binary tree node's encoding process using contexts. The slices are along axis z. The read contour denotes the current node being encoded and the blue contour the image whose 3D contexts is extracted.

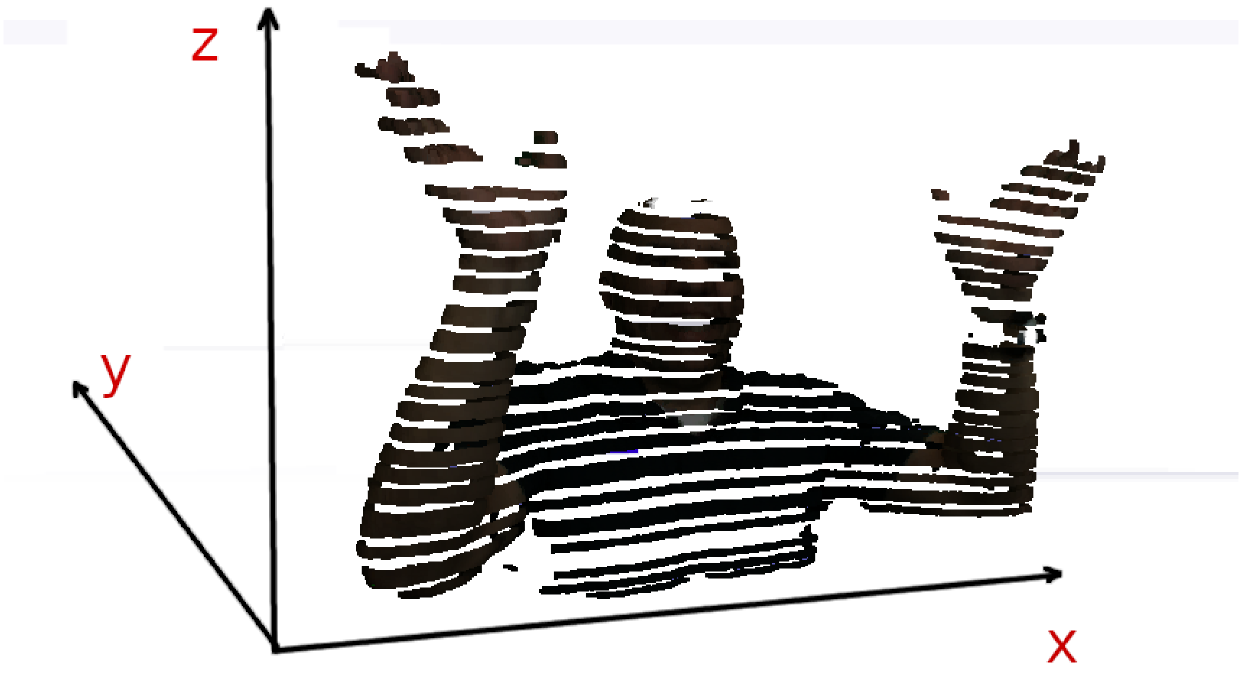


Figure 2.10: An illustrative purpose image depicting slices images from decode - leaf nodes - merged and reconstituting the original point cloud on axis Y.

the data structures design and C++ language syntax and properties choices will be discussed, making more clear the C++ language capabilities and power on developing the S3D algorithm implementation. An additional important topic briefly discussed is the Doxygen documentation of this migrated code, a crucial part of software projects. It will be shown its advantages on IDEs environments and how this documentation was made, along with its resulting document generated.

3 IMPLEMENTED SOLUTION

Given a broad overview on the theory basis that explains the whole mechanism of the S3D algorithm, from the geometry encoding/decoding process until the CABAC using the different contexts depending on the silhouette being encoded, now we will focus on the code implementation of the encoder using the C++ (based on the compiler g++, version c++11). First, we will overview the Matlab project structure and main function modules. Then, with a sufficient understanding and inspiration based on the original code, we will describe the C++ migration project.

3.1 MATLAB PROJECT

Before we describe and have a brief analysis on the Matlab project, it is important to remember some Matlab language's characteristics and qualities compared to C++. First, the Matlab is a language that strongly aims matrix operations and plotting, and user own defined data's structure design. This means that the language's data structure is restricted to few options to user choice. In C++ STD library, instead, there's a plenty built-in data structures options to choose when we implement a specific class data attribute - such as Set, Vector, Array, Queue - offering a more user's freedom choice in favor of most advantageous data structure according to the classes specification. This advantage will make a big difference when we describe the C++ project classes design. Beyond that, all the memory allocation and management is all in charge of the Matlab's interpreter. Due to this, all the data structure's memory occupied during the algorithm's processing which can be freed to other allocations required is not allowed by manually by the user - differently from C++ using `Delete` and `New` command - but depending on the interpreter decision.

With this explained, we will start to analyse the project structure:

```
|_/ArithmeticCoding
|_/Bitstream
|_/Decoder
|_/Encoder
|_/PlyTools
|_/Structs
|_/Utils
```

As we can see above, each folder from this project correspond to a module from the project. Inside each folder, there's the files associated to its respective module, implementing either one specific function or one data structure. This approach presents an advantage: each file composing the module describes a specific function that interacts with the main program flow, allowing short files expressing a important role on the algorithm, and a concise code once all commands from this functions is expressed by its call. On the other hand, when a module presents a lot of functions, the module folder is filled with a plenty of files composing each functionality from the module. For instance, in the *ArithmeticCoding* folder, there's 27 files. Most of them could be clustered in classes or libraries from this main module. In this case, we could create a **BACEncoder(Binary Arithmetic Coder Encoder)** class and a **BACDecoder(Binary Arithmetic Coder Decoder)** class, each one incorporating this module functions as methods and its returns values as attributes - not requiring to get manipulate a constantly used value as return value and passing it as argument for each function, but accessing directly as object's attribute instead. The class absence also exposes some values that should not be manipulated by other program's modules. For example, the point cloud's dimension should not be allowed to be modified by the Arithmetic Coder. This could be simply avoided by the use of **private attributes** on classes.

Moreover, the S3D algorithm's entities such as the Point Cloud and Silhouette images does not have a **interface** and a **encapsulation** abstraction mechanism incorporated on their implementation, but as pure data structures instances. As explained, it turns these data structures during the program flow vulnerable to be accessed and unduly modified by not allowed modules, and possibly complicating a maintenance and modification only on the parts manipulating its data since it's directly accessed over all the program, requiring to modify each access occurrence. Instead of it, using a methods such as **getters** and **setters** facilitates the maintenance process, requiring only to modify the method implementation, in consequence, saving to modify all the access occurrence over the code. We can observe that all these problems could be solved using class implementation.

Another important issue on this project, which is caused mainly by the Matlab's language characteristics, is the data structure choice. When we need for example a **binary tree structure** in order to sort a set of elements such as pixels or voxels, the standard Matlab library does not provide binary tree structure, but needing to implement manually. This bring 2 main problems: a additional code to implementing or import the data structure that may be crucial to the algorithm, and the cost of the user defined implementation of the data structure not be sufficient to explore the language's low levels resources - such as threads - as the built-in functionalities that have directly access to these kind of artifices, specially in proprietary softwares.

3.2 PROJECT STRUCTURE

The original's project overview gives a better idea of the changes that will be observed on the new C++ project. First, we could cite the folders structure. Instead of grouping a set of functions and structures files inside a module folder, each class file incorporates the structures and functions associated with. This design clusters all the modules functionalities inside a file. In one hand, we have a bigger files than the original's project. On the other hand, each module shares variables and functions that is used only inside the module, not anywhere else. Thus, on this work the project's files arrangement is more simple and concise, at the cost of bigger files.

Another important change is the C++ compiled characteristic that forces a code that separates the classes and functions declarations from their definition in different files - header file and source files. This, in addition to simplify the codes analysis and overview by reading only the classes and functions prototypes on the header files, the files size is reduced by separating the declaration code blocks from the definition and implementation code. With this changes understanding, we can now examine the project's characteristics.

Once the project is complex enough by having multiples files and libraries, the compiling process was made using the makefile utility. Thus, the project structure was made so that the compiling process and scripting could be more simplified and pragmatic to test and execute, beyond the fact that the code could be more modularized. Therefore, the project structure is:

```
|_/include
|_/obj
|_/src
|_/test
|_Makefile
|_README.md
```

Where the header files are located in `include`, the source files in `src`, test files on `test` and object files generated from the compiling on `obj`. In each folder we have the main corresponding modules that composes the project: the arithmetic encoder (CABAC) module, configuration module, the data structures module and the encoder/decoder module. Each module is represented by the file name pattern: prefix_file_name, where the prefix denotes the module which it belongs.

The *arithmetic encoder* module covers the mentioned Context Adaptive Binary Arithmetic Coder, dealing with the binary source coding and decoding implementation. It's design follows exactly the implementation explained on section 2.3.4, however storing more contexts than the 1 bit context exemplified. The *configuration module* is responsible for define the S3D parameters that will be used as input on the algorithm execution, as input file, output file and single mode option (although not implemented yet). The *data structures* covers all the abstract structures that will be required to represent each data object used on the algorithm and their essential operations to be used on the main algorithm, such as images, point cloud, voxel and pixel. It's important to mention that this data structures design aimed to be a general API, allowing its use and operation on other coders than S3D. Finally, the *encoder* module is responsible for the silhouette decomposition, applying the corresponding masks to each silhouette node on the binary tree and performing the geometry coding process.

As mentioned, this project was developed by a students and professors team from Universidade de Brasília, where each person focused a specific module or project feature. At this work, the role was mainly focused on design the data structures API required on the development of the S3D algorithm, inspired on the previous matlab work, but with some adaptation. This project followed the Google C++ Style Standard on the variables, functions, namespaces and classes naming, in order to have a more consistent and organized project. More information about this standard is available on the Google Standard's site [20].

In order to verify the functions behaviour and it's correctness, the project classes was developed along with tests cases, making sure that its methods and attributes presents the results expected from different scenarios. The test framework used was GoogleTest by using it's library gtest.h [21]. Thus, these test cases was developed to ensure a correct, consistent and robust code on the data structures developed implementation, allowing a more reliable API not only to this S3D algorithm implementation, but as a library that can be used on other projects.

3.3 C++ MODELING

Here we will describe the design of the data structures used in the algorithm performing and the main motivations that conduced the developed version. It's important to mention that the original Matlab S3D implementation most of the data abstractions from the algorithm was implemented as raw data structures such as matrix or vectors. Thus, in order to fit each data abstraction from the algorithm with the C++ data structures, all the Standard Template Library (STL) pre-built data structures chosen were thought in terms of performance (memory and time), usability by the developers to implement the functions required and attend the design which they were based on. Therefore, besides the language difference between the Matlab to C++ and original data structures migration, this new implementation aims to use and avail all the qualities provided by the C++ features that is useful and advantageous to the algorithm implementation, that includes the STL library. With these choices, the API library provides a consistent module with a good usability, and consequently, helps its use on others modules development that composes the S3D implementation.

It's important to note that this algorithm version does not implement the single mode choice on the encoding, compared to the original algorithm [9]. Besides, the context table are all initialized with values as 1, adapted along the encoding process. The modelling can is ilustrated on Fig. 3.1

3.3.1 Pixel and Voxel

The classes *Pixel* and *Voxel* are the basic classes that is intensively used on the Silhouette Image and Point Cloud data abstraction. Each one shares several operations similarities between them, differing only on the dimension to which they are applied. In the case of Pixel, 2D images, and Voxel, 3D objects, i.e., point clouds.

Being for general purpose, they accept the coordinate system values assume arbitrarily developer defined types, by the use of templates. The coordinate values are represented by an short array, with dimension coinciding with its real dimension.

The most important from the Pixel and Voxel classes are its operations, that allows a more expressiveness and simplified code. The basic algebraic operations such as summing, subtracting, multiplying, dividing, compare (less, greater, less or equal, greater or equal), all of them are overwritten to these classes, allowing to the compiler interpret them as their own operations, and automatically, adapt by itself the C++ pre-built functions to work with these new defined operations. For instance, let's consider a vector of pixels. Using the defined operations of comparison less ($<$) and less or equal ($\leq$), determining as prior compare the first dimension, then the second and finally the third, the C++ sorting algorithm automatically adapts its value comparison with these new comparison mechanism on which it will be based.

Consequently, a significant amount of code implementation and writing is saved by simply defining these atomic operations from these classes. Given that, larger classes that encompass these classes are benefited availing these operations, instead of redefine the operations criteria for each class abstraction application. This will become more clear, on the next classes description. The main functions from these classes are overwriting of basic algebraic operations such as arithmetic operations $+$, $-$, $*$, $/$, and boolean

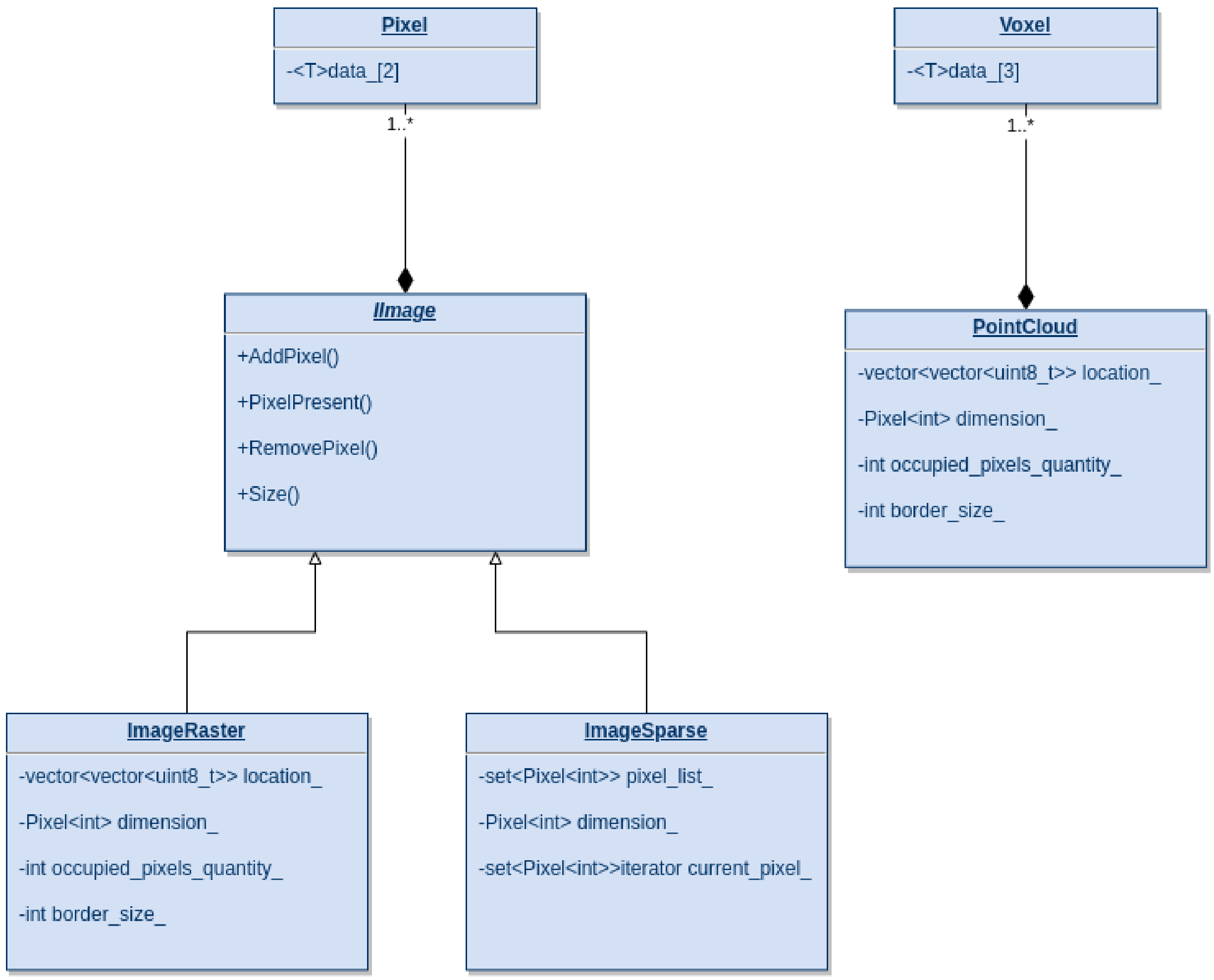


Figure 3.1: S3D data structures classes and it's relations

operations $==$, $<$, $<=$, $>$ and $>=$, which enables the properties described. Other functions are syntactic sugar that describes these functions, such as $+=$ on C++.

3.3.2 IImage

In order to represent the silhouettes that are obtained during the recursive slicing of the point cloud, the general class **IImage** was created. The idea of this class is to represent the general and basic operations that is expected from a silhouette during the algorithm mechanism, which in more practical terms is a bitmap 2D image. These basic operations include add a pixel, remove, and verify occupancy of a specific coordinate from the image.

This general image class, having this basic functions, designs the abstraction that all different image representation on the program should have. Thus, this class acts as a superclass of these derived classes: the ImageSparse and ImageRaster, fig 3.2. Each one of these different implementation of the same abstraction bitmap image has a particular executing performance purpose, such as speed and memory allocation, once this algorithm process thousands of 1024x1024 images. As the program needs only carry the image data and not access operations, thus requiring storage saving and less speed performance, the image ImageSparse representation is used during the program. In the other hand, if there's several image access operations and speed is a priority, only requiring few images - and because of that ocupying less storage -, not requiring memory saving, the ImageRaster is directed for this purpose. In these classes description, this will be explained more clearly. Therefore, the main point of having these subclasses is to have a better performance on the running of the algorithm, availing its best advantages in favor of use less storage and more speed depending on which step of the algorithm these properties are required.

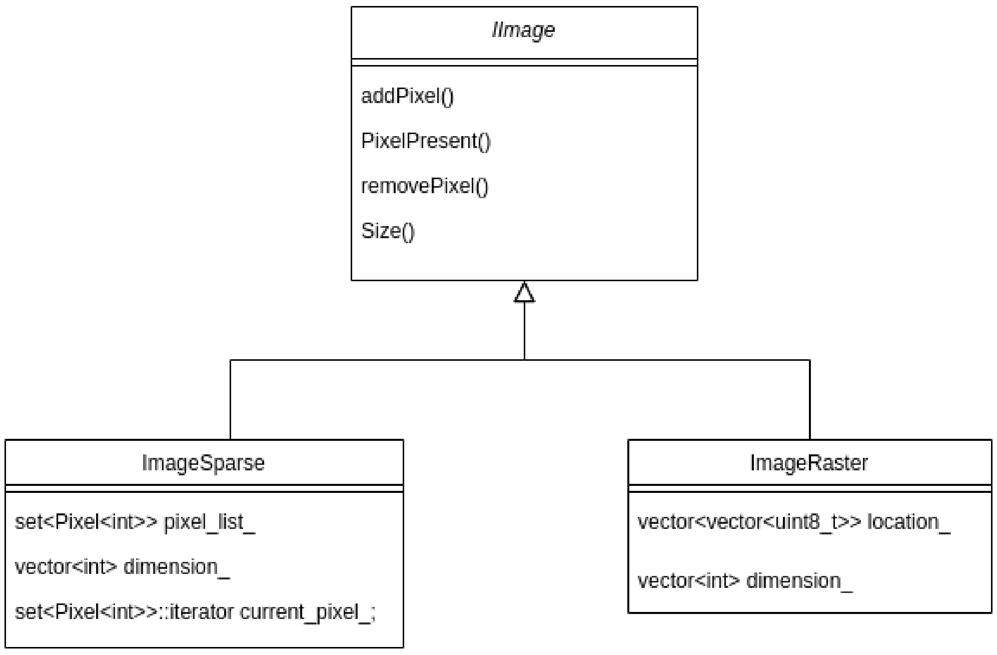


Figure 3.2: The IImage superclass and its subclasses ImageSparse and ImageRaster

This class has no attribute, describing only the templates functions to the images representations. The main functions from the IImage class are listed:

1. `addPixel (int x, int y)`: Receives a (x, y) coordinate and adds a pixel at that coodinate.
2. `removePixel (int x, int y)`: Receives a (x, y) coordinate and removes a pixel at that coodinate.
3. `PixelPresent (int x, int y)`: Receives a (x, y) coordinate and returns a boolean indicating the occupancy at that coordinate.
4. `Size ()`: Returns the image's $Height \times Width$ dimension
5. `NumberOccupiedPixels ()`: Returns the amount of pixels occupied on the image

3.3.3 ImageSparse

The class ImageSparse is the silhouette representation using a Set of Pixels instances, from the class Pixel mentioned, as a data structure of the bitmap image. Before we proceed explain the ImageSparse class mechanism, it's important to have a overview of what a C++ STL's Set data structure is.

The Set is a pre-built data structure from the C++ STL that represent containers that store unique elements following a specific order. This specific order follows the type or class comparison convention defined. This data structure is implemented by a *binary search tree* (BST), ensuring the sorting of elements through performative operations on creation and updating the set, most of them in O $(\log n)$ complexity. At this particular case of Set of Pixels, the comparison of Pixel is predefined on the class definition, prioring the 1st and then 2nd value as comparison. Thus, the library will automatically sort the Pixels following this convention.

Having a basic notion of set, we can now understand the mechanism behind the ImageSparse class. Considering that this class stores a set of Pixels, where each Pixel represent a coordinate on a 2D plan, the idea is store only the information that we need from the image. If the image is a bitmap representing basically whether the pixel is occupied or not, each pixel from the image can represent only two cases: a occupied pixel, or a empty pixel. Besides, in the algorithm application the images will not be processed all the time, instead only in required moments. Thus, if the algorithm necessity is only store the image and not access it content, and beyond that, actually requiring only the minimum information capable to reconstruct the original image, it's convenient to store only the occupied pixels from the silhouette. Using this approach, the memory usage will be significantly reduced, in view of the fact that a great percentage of needless pixels will be discarded. That's why it is a sparse version of the image.

This memory performance allied to the SLT Set advantage of sorting the Pixels automatically and ensuring unique elements makes the pixels access and arrangement of the image more consistent and structured for further usage on the algorithm. One drawback from this representation is the elements - Pixel - access, once the operations upon this data structure is a BST operations, requiring a time complexity around $\log n$ for the value searching on the tree, where n is amount of data. Consequently, the purpose of this representation is more storing than processing.

The ImageSparse class, being a sublclass, has the same functions as the IImage class, by the use of its own Set implementation. Its main attributes are:

1. `pixel_list_`: A set of all occupied pixel's coordinates.
2. `dimension_`: Height and Width of the image silhouette
3. `current_pixel_`: A iterator pointing to the current pixel on the set traversing

The additional functions are related to operations on its iterator on the pixels set:

1. `CurrentPixel ()`: Returns the `current_pixel_` iterator. $O(1)$ time complexity
2. `NextPixel ()`: Moves the `current_pixel_` one ahead. If it's the last pixel, returns to the beginning. $O(1)$ time complexity
3. `PreviousPixel ()`: Moves the `current_pixel_` one back. If it's the first pixel, goes to the last pixel on the set. $O(1)$ time complexity

3.3.4 ImageRaster

Explained the ImageSparse representation, now we will describe the ImageRaster representation. This class is a classical 2D bitmap matrix representation of the silhouette, using the STL Vector data structure. Thus, the class holds the data about the image by using `vector<vector<uint8_t»`, i.e, a vector of vector(2D matrix) of 8 bits integers, and the image's dimension.

Once this representation stores the whole image, containing each pixel from the image, this consumes more runtime memory usage than the ImageSparse. But in the other hand, being each pixel coordinate associated with two index (y, x) directly, its access can be directly made through the dereference provided by the indexing, don't requiring a searching. This direct access is of time complexity constant $O(k)$, which gives more speed performance on the image processing. Using more storage but saving time are properties that best fits on the case of perform all the encoding operations on the silhouette, seeing that this algorithm step needs thousands of access on the image's pixel. For instance, let's consider a given operation that needs to run over all the pixels from the image. For the ImageSparse case, we would have a given ρ percentage of occupied pixels of the whole image with n data. If each operation on each pixel is given by the time complexity $O(\log n)$, we would have a resulting time complexity around $O(\rho n \log n)$, which is equal to $O(n \log n)$. Now considering a ImageRaster image representing the same amount of data n. If each operation on the pixel is $O(k)$, we have a total complexity of $O(kn)$, which is equal to $O(n)$. Thus, we can conclude that the ImageRaster have a better performance than the ImageSparse, being a better option on the image processing steps.

The ImageRaster being a 2D matrix representation of the silhouette, it contains the data related to this matrix and its description:

1. `location_`: The 2D matrix, where each element is 1 byte integer. Represents the silhouette image.
2. `dimension_`: Describes the image silhouette's Height and Width.
3. `occupied_pixels_quantity_`: Describes the image silhouette's Height and Width.

Its functions template is equal to the IImage description, differing only in the data structure based on the 2d matrix, and consequently, its implementation.

3.3.5 PointCloud

Finally, we have the Point Cloud class. This class is very similar to the ImageSparse, being, instead of a Set of Pixel, a Set of Voxels, based on the same data structure Set from C++ STL. This implementation choice was motivated by the reason that the point cloud, at the S3D algorithm, is not much processed or manipulated on the encoding, being required only to obtain the silhouettes from it. Once the knowledge of the voxels occupied on the volume is sufficient to generate the silhouette images, just storing these voxels attend to the algorithm purpose. By the same way as on ImageSparse, the voxels is sorted following the priority on 1st, 2nd and then 3rd dimension on the Set.

An important method from the Point Cloud class that is crucial on algorithm mechanism is the silhouette generation given an interval from the volume and axis along which the image is projected. Another important method is recover voxels from a given silhouette, getting all the pixels from the image and inserting on a specific plan from the point cloud, and specific axis coordinate where this plan locates, as depicted in Fig. 3.3.

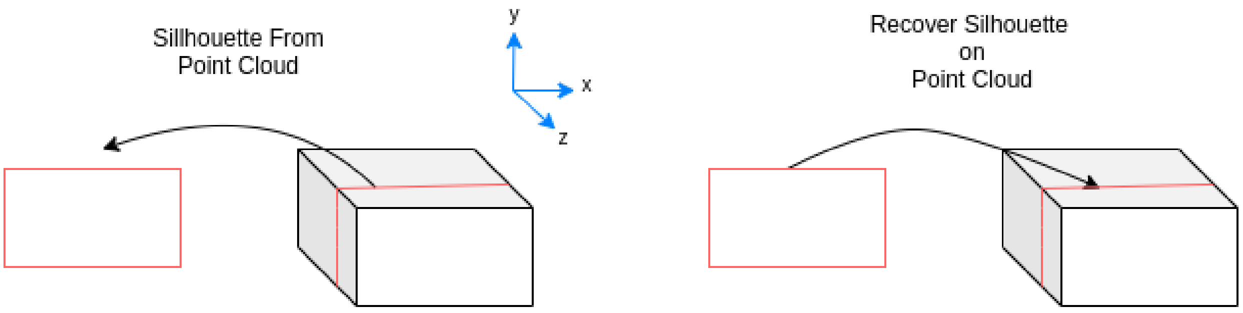


Figure 3.3: Point Clouds methods: Slice a silhouette from point cloud and recover silhouette slice on point cloud.

The Point Cloud class main attributes are similar to the previously described Image classes:

1. `voxel_list_`: It's the 2D matrix integer matrix, storing the values 0 or 1 denoting the pixels occupancy.
2. `point_cloud_voxel_count_`: Amount of voxels occupying the volume.
3. `dimension_`: Describes the Height, Width and Depth from the point cloud volume

And its main functions, similar as well:

1. `addVoxel (int x, int y, int z)`: Adds a voxel at the specified coordinate. $O(\log n)$ time complexity
2. `removeVoxel (int x, int y, int z)`: Removes a voxel at the specified coordinate. $O(\log n)$ time complexity
3. `VoxelPresent (int x, int y, int z)`: Verifies the occupancy at the given coordinate. $O(\log n)$ time complexity
4. `Size ()`: Returns the dimension of the point cloud. $O(k)$ time complexity
5. `SilhouetteFromPointCloud (int slice_start, int slice_stop, Axis axis)`: From a given axis, slices a point cloud in a given range, and from this slice projects a silhouette along the same axis. $O(n \log n)$ time complexity
6. `RecoverVoxelsFromSilhouette (Axis axis, int coordinate, ImageRaster* image)`: On a given axis and coordinate at this axis, inserts a image's silhouettes. $O(n \log n)$ time complexity

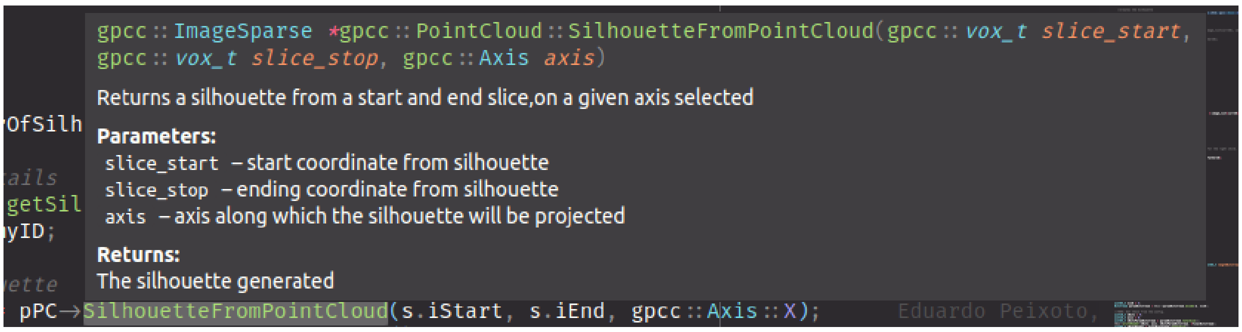
3.4 DOCUMENTATION

An important concern on this project development is to ensure the readability, and the same time, expressiveness that the data structures API provides to the user. These two properties was thought aiming to encourage further works and optimizations on this code by the team and new incoming members. Thus, all the project design was focused on attending these good practices, from the functions, classes, namespaces, and its attributes names to the code's comments. Another crucial artifact on the software development is the documentation.

On the case of this project, all the documentation was made through the header files comments, describing the functions declaration and prototypes. These file comments was thought and written using the Doxygen standard. This approach has 2 advantages on the developing. The first is advantage is the Integrated Development Environments (IDE) facilities. The IDEs usually uses these comments on the header files as a reference to the codes documentation, automatically providing to the user the description of the functions when its needed for instance. In the Visual Studio Code, we have the following function declaration written as:

```
//! Silhouette making
/*
 * Returns a silhouette from a start and end slice,
 * on a given axis selected
 *
 * \param slice_start start coordinate from silhouette
 * \param slice_stop ending coordinate from silhouette
 * \param axis axis along which the silhouette will be projected
 * \return The silhouette generated
 */
ImageSparse *SilhouetteFromPointCloud (
    vox_t slice_start, vox_t slice_stop, Axis axis);
```

Using this comment pattern, Qt style [22], the IDE automatically indicates to the user the definition described to this function as depicted in Fig. 3.4.


```
gpcc::ImageSparse *gpcc::PointCloud::SilhouetteFromPointCloud(gpcc::vox_t slice_start,
gpcc::vox_t slice_stop, gpcc::Axis axis)
```

Returns a silhouette from a start and end slice,on a given axis selected

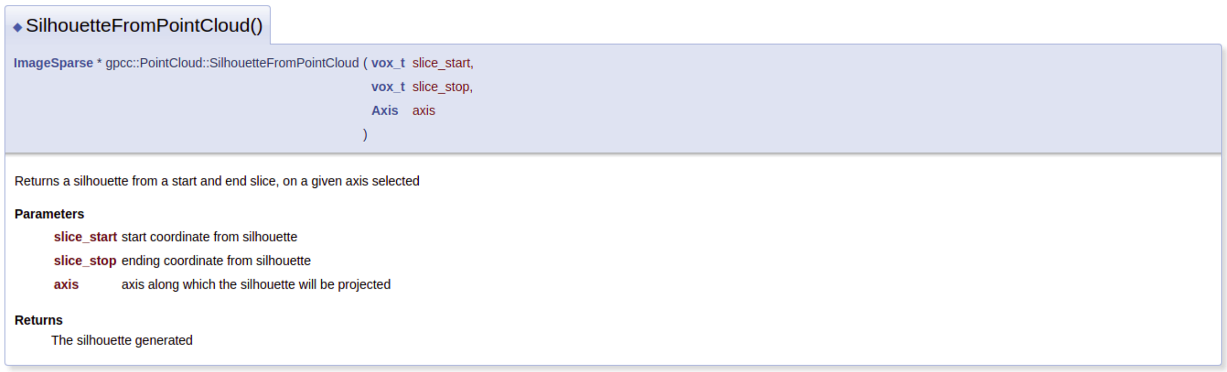
Parameters:
`slice_start` – start coordinate from silhouette
`slice_stop` – ending coordinate from silhouette
`axis` – axis along which the silhouette will be projected

Returns:
The silhouette generated

```
pPC->SilhouetteFromPointCloud(s.iStart, s.iEnd, gpcc::Axis::X);     Eduardo Peixoto,
```

Figure 3.4: Comment documentation on Visual Studio Code

The second advantage is the Doxygen parser on these documenting comments. When the Doxygen is run, these documenting comments syntax above the functions declaration is identified and a document describing all the codes documentation is generated. A variety of document types, styles and choices can be made on the documentation generating. For example, a simple HTML front page folder can be generated on this process. The above code comment generates the HTML doxygen documentation as in Fig.3.5. The classes inheritance and relations is automatically illustrated as in 3.6.



◆ SilhouetteFromPointCloud()

ImageSparse * gpcc::PointCloud::SilhouetteFromPointCloud (vox_t slice_start,
vox_t slice_stop,
Axis axis
)

Returns a silhouette from a start and end slice, on a given axis selected

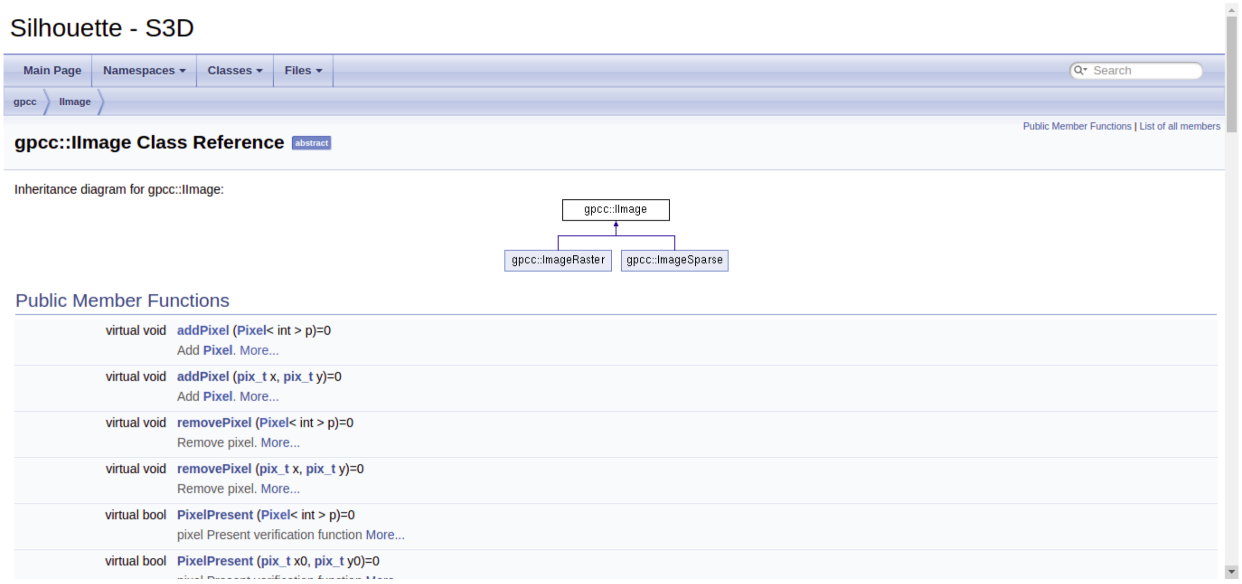
Parameters

slice_start	start coordinate from silhouette
slice_stop	ending coordinate from silhouette
axis	axis along which the silhouette will be projected

Returns

The silhouette generated

Figure 3.5: Doxygen HTML documentation generated



Silhouette - S3D

gpcc::IImage Class Reference

Inheritance diagram for gpcc::IImage:

Public Member Functions

virtual void	addPixel (Pixel< int > p)=0 Add Pixel. More...
virtual void	addPixel (pix_t x, pix_t y)=0 Add Pixel. More...
virtual void	removePixel (Pixel< int > p)=0 Remove pixel. More...
virtual void	removePixel (pix_t x, pix_t y)=0 Remove pixel. More...
virtual bool	PixelPresent (Pixel< int > p)=0 pixel Present verification function More...
virtual bool	PixelPresent (pix_t x0, pix_t y0)=0

Figure 3.6: Classes inheritance represented in the doxygen documentation

CONCLUSION

Here were covered all the data structures and S3D algorithms abstractions API implementations using classes and the C++ STD library. We could cite the classes migrated from the Matlab as well as the new classes created during the migration process considering its performance advantages acquired on the data structures choices, such as IImage, ImageSparse and ImageRaster. Moreover, we have seen the C++ function overwriting and the possibility of a more flexible code developing using this feature, such as sorting objects from a Set by overwriting the comparison operators.

Now that all the code implementation was explained and its design choices detailed, it's important to remember that all this code was thought aiming a performative version of the S3D algorithm. This means a algorithm implementation version with lesser execution time and similar compression rate, since it's a migrated code. Thus, the next chapter we will assess it's performance results values and compare with the original implementation, and other GPCC (Point cloud Coders), so that verifying if the project's main goals will be achieved by this work.

4 RESULTS

Now that the C++ project was described and the algorithm's background mechanism explained, the implemented project is assessed in order to examine its performance. It's important to remember the project's motivations that determines its main goal when it was originally planned: the requirement of a **similar performative algorithm** as the S3D Matlab implementation, since it's a reproduction of the original implementation, and a **time outperform over the original**, allowing a further study and exploration on its complexity and characteristics that may optimize its results.

It's important to note that this S3D algorithm version has some technical differences that produces different bitstreams from the original. One of them is the decimal number rounding process from Matlab that differs from the C++ on the arithmetic encoding. Another difference is the context initialization, where on the new implementation does not preprocess the silhouettes but initialize all contexts as 1. Despite of this fact, the algorithm performance is expected to present a similar result, considering that this differences doesn't modify the algorithms original proposal of combining the silhouette decomposition with the CABAC encoder.

In order to assess this work project, a set of public dynamic point clouds available in ply files was used as input to the program. The dataset used was the "Microsoft Voxelized Upper Bodies - A Voxelized Point Cloud Dataset", from JPEG's plenodb [16]. Given this dataset as input, a set of measures of the program's performance was collected, examined and compared to the original implementation and other codecs. The selected point clouds were those with 9 bits ($512 \times 512 \times 512$ dimension): Andrew9, David9, Phil9, Ricardo9, Sarah9. Each one represents a dynamic Point Cloud with 318, 216, 245, 216, 207 frames, respectively.

The measures concerning the Matlab and C++ execution time and rate was performed on a Desktop computer with a **Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz, 32.0 GB Installed RAM, x64-based processor, running on OS Windows 10 Education Edition**, as listed on the Tables 4.1, 4.2, 4.3, and plots 4.2, 4.3, 4.4, 4.5, 4.6. The program profiling time data was collected in a notebook **Intel® Core™ i7-7500U CPU @ 2.70GHz × 4, 7.6 GiB, x64-based processor, running on OS Linux distro Ubuntu 18.04 LTS**, as in the tables 4.4, 4.5, 4.6, 4.7.

4.1 GATHERED RESULTS

Before we go further on the results analysis, briefly we will explain how the performance data from the C++ implementation execution was collected. Consider the dynamic point clouds from the Microsoft Voxelized Upper Bodies. For each one, the first 20 frames were picked, encoded, then, decoded. This process was repeated 5 times, giving to each point cloud 100 measures. All the information concerning compression rate over each axis, the time spent on each execution was collected and exported on a **csv format** at each encoding-decoding measure. Thus, each frame having 5 repeating measures, we excluded the outliers measures and computed the mean between the 3 left measures. With this process, each frame has the average time execution and rate data, which is used on the comparison process on the following results depicted.

First, the analysis will be on the C++ project rate compression performance, measuring its bit per occupied voxel results. Its values is compared with usual and well known GPCCs in Table 4.1. The data result from the C++ S3D was collected in the first 20 frames, for each axis and selecting its best rate, and the other GPCCs results on 200 frames. We can observe that the S3D performance, even though not outperforming as in the Peixoto's work [9], presents a result near as the MPEG's TMC13 GPCC. This can be explained by the fact that the amount of frames rate compression measures from the C++ S3D is less than the others, being 10 times less, making the average rate not be precise as possible. Moreover, the encoding was performed without the single mode, which could present a better performance. This results shows that the C++ S3D can have optimized performance, nearly matching and even outperforming *state-of-art* GPCCs.

Table 4.1: Compression algorithms rates comparisons:

Sequence	**Average Rate(first 200 frames)**						**Average Rate (first 20 frames)**			
	Octree	P(PNI)	P(Full)	BDC	TMC13	S3D(Full)	P-X	P-Y	P-Z	S3D Best Rate
Andrew	2.58	1.83	1.36	1.70	**1.14**	**1.12**	1.23	1.24	**1.21**	**1.21**
David	2.62	1.77	1.33	1.68	**1.08**	**1.06**	1.23	1.23	**1.23**	**1.23**
Phil	2.64	1.88	1.43	1.71	**1.18**	**1.14**	**1.25**	1.29	1.26	**1.25**
Ricardo	2.59	1.79	1.34	1.66	**1.08**	**1.04**	1.11	1.13	**1.11**	**1.11**
Sarah	2.61	1.79	1.30	1.64	**1.07**	**1.07**	1.22	1.23	**1.20**	**1.20**
Average	2.61	1.81	1.35	1.68	**1.11**	**1.08**	1.21	1.22	**1.20**	**1.20**

Table 4.2: Matlab and C++ S3D implementation comparisons

Sequence	**Rate(BPOV)**			**Encoding Time(s)**			**Decoding Time(s)**		
	Matlab	Matlab Init.	C++	Matlab	Matlab Init.	C++	Matlab	Matlab Init.	C++
Andrew	1.31	1.23	1.21	28.20	27.58	5.78	29.28	29.52	3.30
David	1.32	1.24	1.22	36.53	32.40	6.86	40.75	36.37	3.63
Phil	1.34	1.27	1.25	40.91	39.24	7.31	43.35	42.40	3.57
Ricardo	1.21	1.12	1.10	24.38	23.78	3.14	27.07	26.83	2.28
Sarah	1.30	1.21	1.20	32.18	30.08	6.58	34.40	32.24	3.51

The second comparison assess the S3D compression rate and time performance between the different versions, as depicted on Table 4.2. The **Matlab** is the original S3D implementation published, **without the single mode**. The **Matlab Init.** is the original version with modification on the context Tables, initialized with 1(in array representation, as [11]). And finally, the C++ is the migrated implementation using initialized context as 1 as well. The Matlab and C++ times were obtained following the mechanism explained before. But the Matlab Init. time measures was collected only one time per frame, not measured in average. These result values shows that the Matlab compression rate is worse the Matlab Init and C++, indicating that the context initialization with 1 produces a optimized performance. And comparing the Matlab Init.

Table 4.3: C++ migration S3D gains over original implementation

Sequence	Rate Gain(%)		Encoding Time(%)	Decoding Time(%)
	Matlab Init.	C++	C++	C++
Andrew	-6.06	-7.63	-79.50	-88.73
David	-7.58	-7.58	-81.22	-91.09
Phil	-6.72	-6.72	-74.82	-91.76
Ricardo	-9.09	-9.09	-87.12	-91.58
Sarah	-7.70	-7.69	-79.55	-89.80

Table 4.4: Encoding implementation profiling

Function	% time	cumulative seconds
CodecParameterBitstream::initContexts(unsigned int)	20.28	2.47
tinyply::...::read_header_property(std::istream&)	10.63	3.77
tinyply::...::request_properties_from_element(...)	4.02	6.46
gpcc::PointCloud::Load(std::__cxx11::basic_string<char,...)	5.83	5.36
gpcc::ImageRaster::CompareToRaster(gpcc::ImageRaster*)	5.01	5.97
tinyply::...::make_property_lookup_table()	5.01	5.97
gpcc::PointCloud::PointCloud(std::set<gpcc::Voxel<int>,...)	3.20	7.70
tinyply::PlyFile::get_info[abi:cxx11]())	3.20	8.09
gpcc::ImageRaster::ImageRaster(std::__cxx11::basic_string<char,...)	3.12	8.47

Table 4.5: Encoding implementation profiling

Function	self seconds	calls
CodecParameterBitstream::initContexts(unsigned int)	2.47	1309
tinyply::...::read_header_property(std::istream&)	1.30	1113237908
tinyply::...::request_properties_from_element(...)	0.49	379487671
gpcc::PointCloud::Load(std::__cxx11::basic_string<char,...)	0.71	186881950
gpcc::ImageRaster::CompareToRaster(gpcc::ImageRaster*)	0.61	300572682
tinyply::...::make_property_lookup_table()	0.61	300572682
gpcc::PointCloud::PointCloud(std::set<gpcc::Voxel<int>,...)	0.39	325776962
tinyply::PlyFile::get_info[abi:cxx11]())	0.41	318585181
gpcc::ImageRaster::ImageRaster(std::__cxx11::basic_string<char,...)	0.38	152041784

and C++ rates, they share similar performance values, having a small difference. This is explained considering that, even though both shares the same algorithm process, the functions that involves rounding and truncation differs between the languages. These small difference can imply in a big modifications on the arithmetic encoding step, which uses these operations. Following, the encoding time and decoding time results allows to conclude that the C++ implementation saves significantly more time execution than the Matlab implementation, considering the compiled mechanism from C++ language compared to the interpreted mechanism from language's as Matlab, which mostly spents more execution time. Given the previous measures, the rate compression and encoding-decoding time gains is more clearly shown on

Table 4.6: Decoding implementation profiling

Function	% time	cumulative seconds
CodecParameterBitstream::initContexts(unsigned int)	19.75	2.41
tinyply::...::read_header_property(std::istream&))	11.25	3.78
tinyply::...::request_properties_from_element(...)	7.23	4.66
gpcc::PointCloud::Load(...)	5.83	5.37
gpcc::ImageRaster::CompareToRaster(gpcc::ImageRaster*)	5.01	5.98
tinyply::...::make_property_lookup_table()	4.02	6.97
gpcc::PointCloud::PointCloud(...)	3.20	7.36
tinyply::PlyFile::get_info[abi:cxx11]() const	3.20	7.75
gpcc::ImageRaster::ImageRaster(...)	3.12	8.51

Table 4.7: Decoding implementation profiling

Function	self seconds	calls
CodecParameterBitstream::initContexts(unsigned int)	2.41	1309
tinyply::...::read_header_property(std::istream&))	1.37	1113237908
tinyply::...::request_properties_from_element(...)	0.88	232763907
gpcc::PointCloud::Load(...)	0.71	186881950
gpcc::ImageRaster::CompareToRaster(gpcc::ImageRaster*)	0.61	294833409
tinyply::...::make_property_lookup_table()	0.49	379487671
gpcc::PointCloud::PointCloud(...)	0.39	325776962
tinyply::PlyFile::get_info[abi:cxx11]() const	0.39	238791114
gpcc::ImageRaster::ImageRaster(...)	0.38	152041784

Table 4.3. Here, we can observe that execution time could be reduced at least 79.50%, and a maximum optimization of 91.58%, in the case of decoding. Therefore, a significant amount of time optimization to the S3D was brought from the Matlab to C++ migration.

At the plots 4.2,4.3,4.4, 4.5, 4.6, it represented the compression rate over the 20 frames from each point cloud. Each axis plot share similar shapes, indicating that there are certain frames that are naturally more difficult to compress than others, as we can see in david 4.4. This frames represents those that have many different movements from previous, difficulty that makes less efficient the context usage. Thus, we can conclude that independent of the chosen axis, the variability pattern of the performance of the algorithms compression rate according to the frame will remain the same. Besides, the black line representing the best rate cross the lowest point from the three axis curves.

The last analysis will be about the implementation components time influence analysis on the entire application. The profiling were performed over the encoding and decoding process, separately. The program used was the **gprof**. The results on encoding are listed on 4.4 and 4.5. These results shows that most part of the time spent on the process is from the ply file parsing to the point cloud data structure, not just because its self time but also the amount of its call. The most time expensive consumes 10% of the total execution time. Following we have the `initContexts` - which initializes the context tables - point clouds operations, such as `PointCloud::Load` - function that from a ply file generates a Point Cloud instance - and its constructor, and Image related functions as `CompareToRaster` - image comparison function - and its constructor.

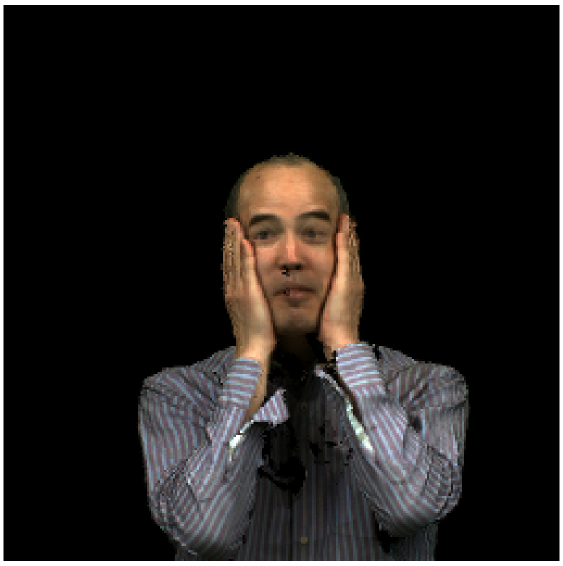
Following we have the decoding profiling, Table 4.6 and 4.7. From the functions listed, we can infer that the classes and functions that spend more time are the same as from the encoding, differing on the percentage usage from the total execution time. And also, the main contributions to this result is the self time and amount of calls of each function, which is around hundred million times. One possible further work in order to solve this time performance is first analyse if the choice of Set as point cloud representation is appropriate and advantageous compared to the time inefficience on atomic operations as voxel access, considering other data structures alternatives. Another possible optimization is on the ply file parsing, which is on the charge of a 3rd party library, opening an opportunity of a development of a own implementation of parser. Futhermore, the image comparing function is another that is worth of refactoring, considering it's inneficient comparison as observed on the results on the tables 4.4 and 4.6.

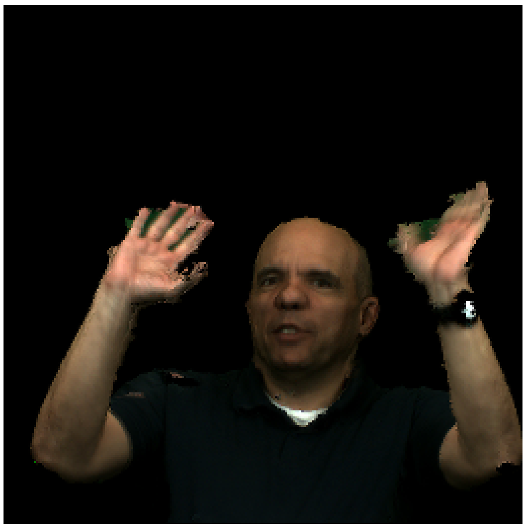
CONCLUSION

This chapter has shown the performance results from the C++ S3D version and the time optimization obtained from the language migration process. Its compression rate, even though not performative as the Peixoto's S3D work, shares values that is close to the MPEG's TMC13 GPCC. The lack of efficiency in comparison with the original work is resulted by the fact that this version do not uses the single mode, and its measurements is not precise as the codec measurements since its less samples amount considered. Furthermore, these measurements indicates the significant influence that the Context Initialization makes on the final result, as depicted in table 4.2. Using this approach, a improvement observed of 1.30 to 1.20 bpov is reachable. We assessed as well the axis choice influence over the final result. Although depending on the point cloud a specific axis brings better results, the variation on the frame compression rate efficiency does not change between them. Thus, the axis choice will not optimize over the macroscopic rate perspective, but influence on each frame compression value, making it more efficient at each frame. Finally, a program profiling was analysed over the encoding and decoding. As concluded, the main classes that influences on the total execution time is the codec context initialization, ply parser, Point Cloud and Image Raster operations. These results reviewed on this project goals and possible optimizations that can be made will be concluded and deliberate in the next chapter.

(a) Andrew9 point cloud

(b) David9 point cloud


(c) Phil9 point cloud


(d) Ricardo9 point cloud

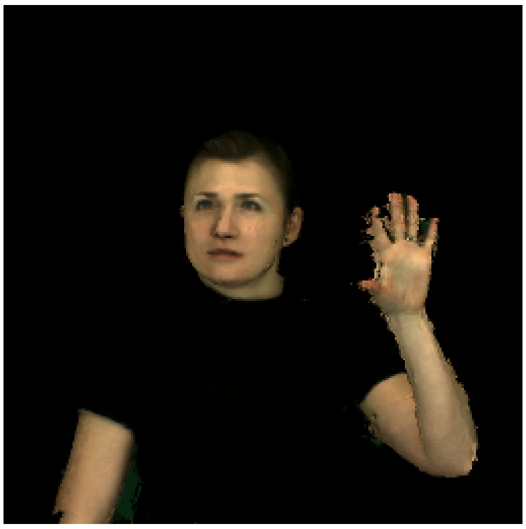

(e) Sarah9 point cloud

Figure 4.1: All point clouds used as input to assess the C++ S3D performance

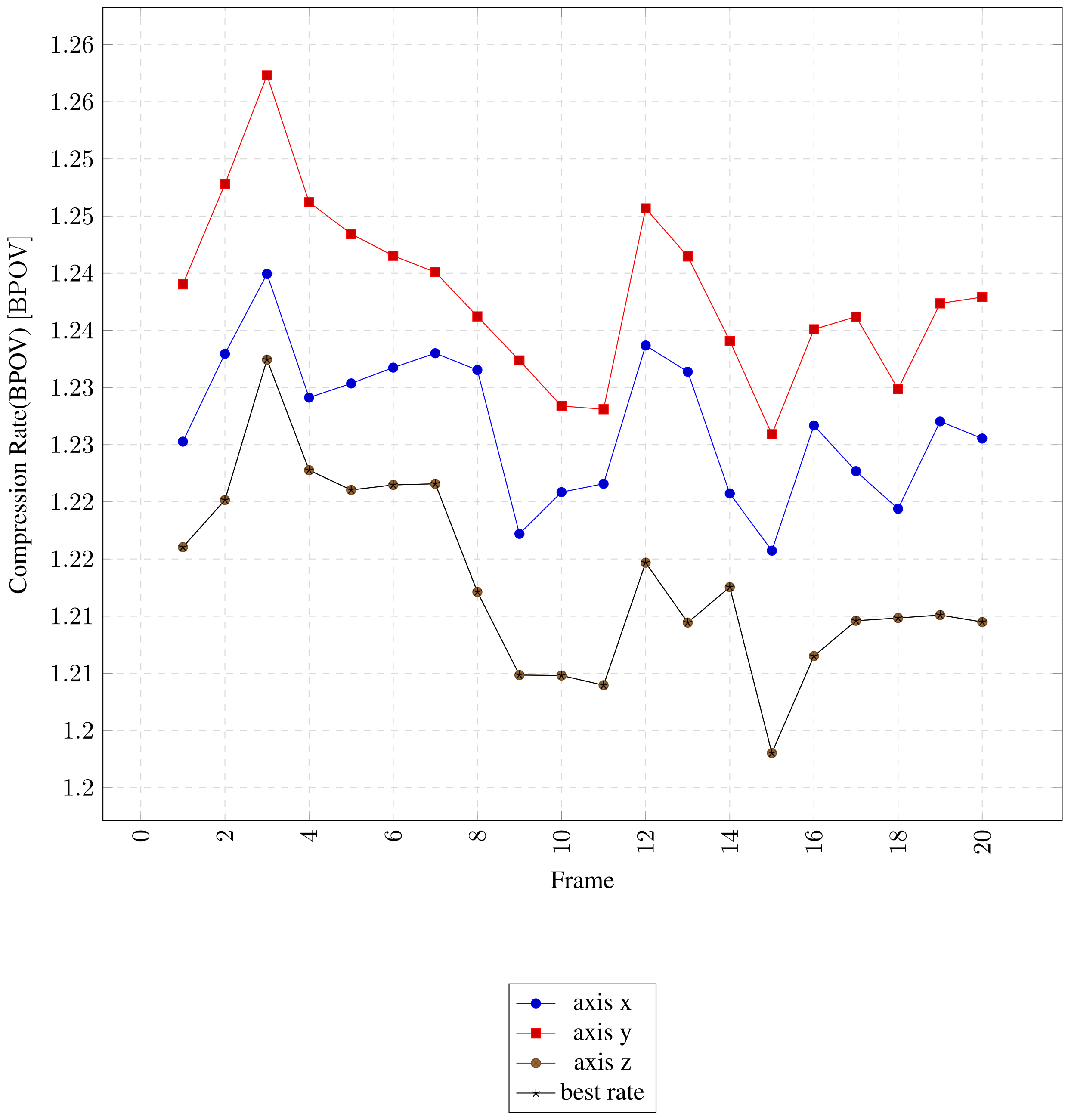


Figure 4.2: Andrew compression rate per frame.

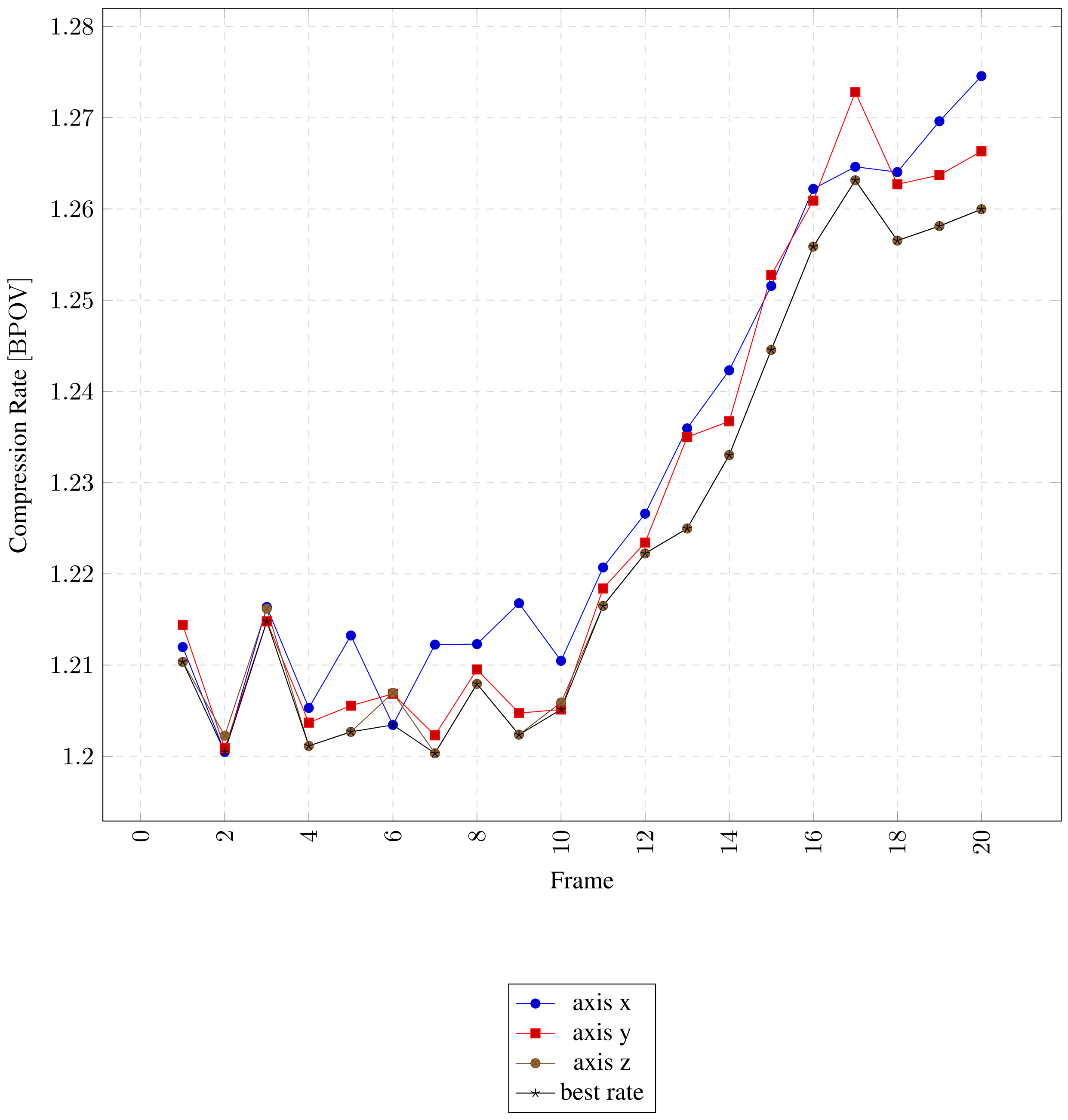


Figure 4.3: David compression rate per frame.

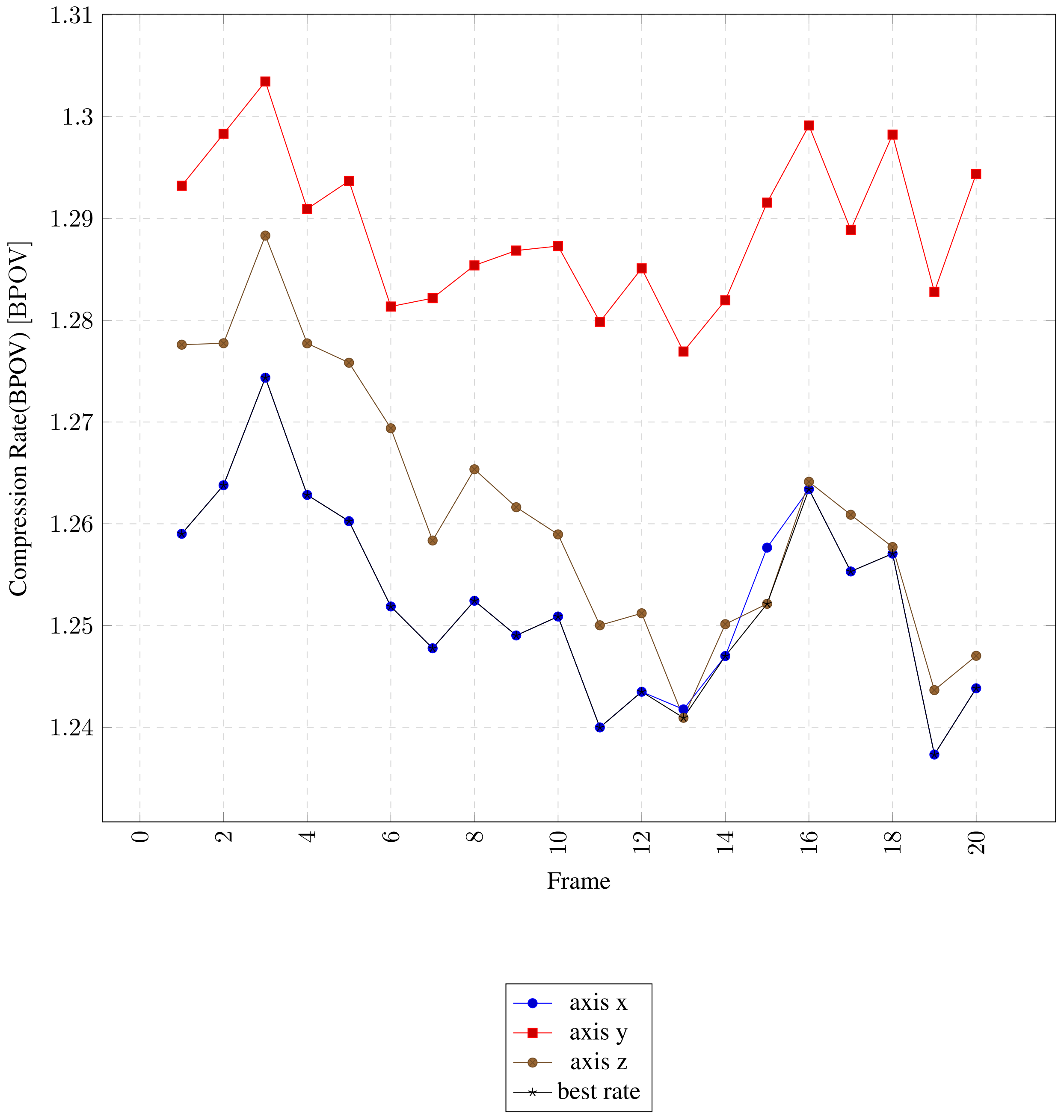
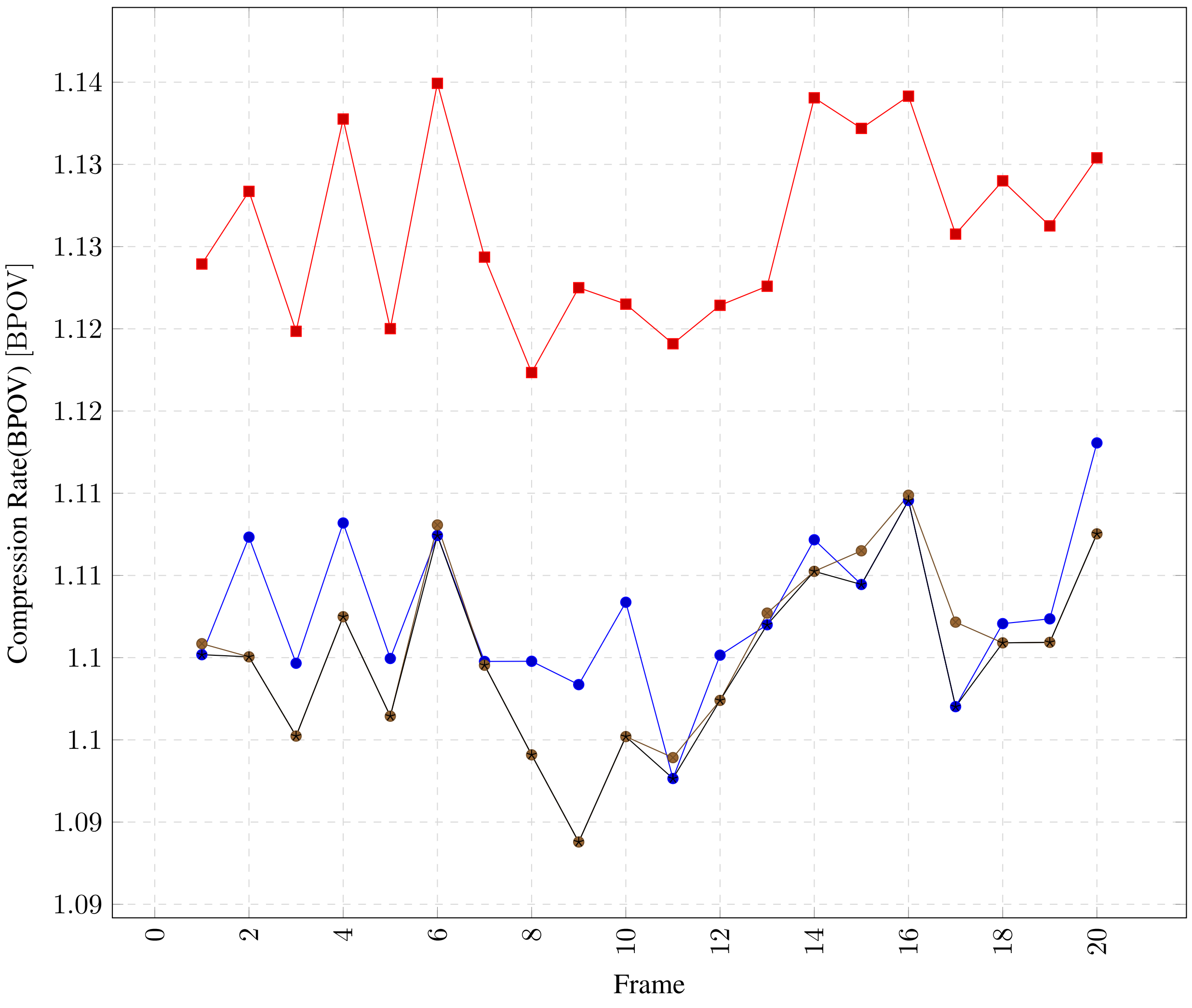


Figure 4.4: Phil compression rate per frame.



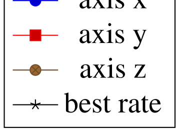


Figure 4.5: Ricardo compression rate per frame.

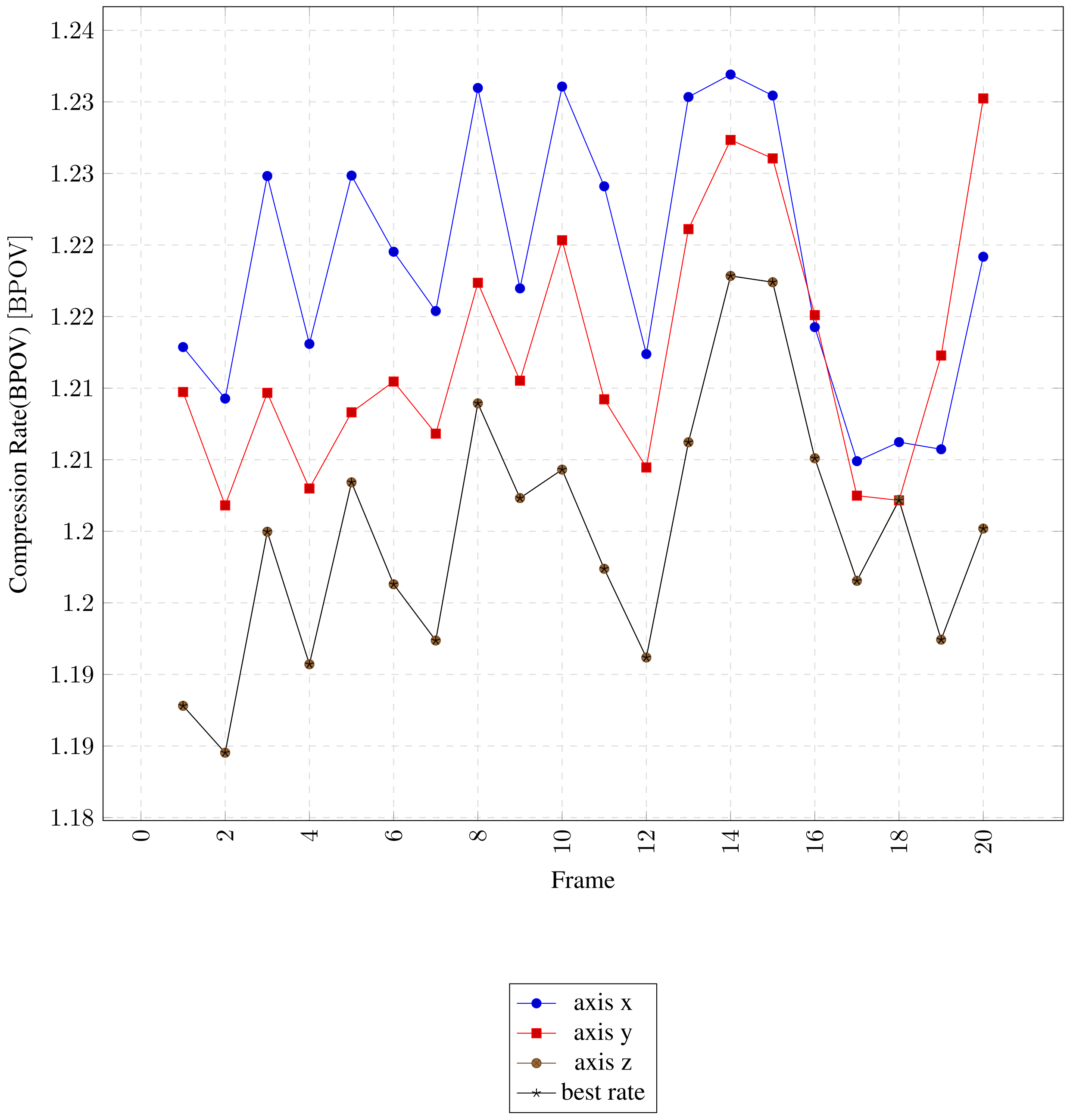


Figure 4.6: Sarah compression rate per frame.

5 CONCLUSIONS

The results chapter has shown the C++ S3D performance compared to other codecs and with its previous version on Matlab. Regarding the other codecs, this project's work presented an outperforming result on the compression rate, and a close performance to the MPEG's TMC13 algorithm and S3D Matlab version. On the other hand, in relation to the Matlab's S3D the new version presented a significantly outperforming performance when the time execution is compared. Both results - similar compression rate and outperforming time execution - verifies the project's goals, indicating the accomplishment on creating a more performative version of the algorithm. It's important to mention the comparison between the S3D versions, which differs the context initialization approach. This analysis showed that initializing the contexts with 1 values makes a more performative algorithm than the not initialized, a result that can guide the successor versions of the algorithm approach.

From these analysis, some issues observed worth a deeper study and testing on further works. These issues concerns possible new features and approaches on the algorithm, and optimization work as well. Moreover, remeasures is also interesting on a deeper analysis on this work, considering that the amount of frames is less than other works measures, as in the Peixoto's work [9]. The first possible and important further work on the project is asses the algorithm performance using a single mode approach as reproduced on the original's S3D proposal. The Matlab and C++ versions comparison on the compression rate indicates, but is not sufficient to infer, that the single mode can optimize the algorithm. Thus, new measures using this approach are important result to answer these hypothesis. Another important possible work is get more measures from the new migrated S3D implementation, around 200 frames as shown in the results chapter, in order to measure more precisely its performance. An interesting proposal not covered on this work is the possibility of using the multi threading on functions and process that can be executed independently. A deeper analysis on this kind of programming approach applied on the S3D worth a deeper exploration on the S3D algorithm's potential. Finally, given the program profiling time measures, the parser, point cloud, image classes and the Context initialization are functions and data abstractions that worth refactoring and optimization process, thus reducing the total execution time.

6 APPENDIX A

This chapter covers the Arithmetic Coder section's 2.3 examples which illustrates how the coding and decoding algorithm works, making more clear the steps involved in each algorithm explained.

6.1 ARITHMETIC CODING

Lets consider an encoding for the sequence with probabilities described in 6.1.

And its important to remember that $X(a_i) = i$. Suppose we want to encode the sequence $\mathbf{x} = (a_3 a_3 a_1 a_2)$.

Table 6.1: Example 1 - Symbols probability table

Symbol	Probability
a_1	0.8
a_2	0.02
a_3	0.18

From the probability model, we know that:

$$F_X(k) = 0,\ k \leq 0,\ F_X(1) = 0.5,\ F_X(2) = 0.9,\ F_X(3) = 1,\ F_X(k) = 1,\ k > 3$$

By the equations 2.10 and 2.11, we performs the bounds computation sequentially to obtain the tag interval. Initially, $u^{(0)}$ is 1 and $l^{(0)}$ is 0. The first element of the sequence is a_3. Thus:

$$\begin{aligned}
&\mathbf{symbol} = (a_3),\\
&l^{(1)} = 0 + (1-0)0.9 = 0.9\\
&u^{(1)} = 0 + (1-0)1.0 = 1.0
\end{aligned}$$

the tag is then contained in $[0.9, 1.0)$. Following the next element a_3, we have:

$$\begin{aligned}
&\mathbf{symbol} = (a_3),\\
&l^{(2)} = 0.9 + (1-0.9)0.9 = 0.99\\
&u^{(2)} = 0.9 + (1-0.9)1 = 1.00
\end{aligned}$$

similarly, to a_1:

$$\begin{aligned}
&\mathbf{symbol} = (a_1),\\
&l^{(3)} = 0.99 + (1-0.99)0.0 = 0.990\\
&u^{(3)} = 0.99 + (1-0.99)0.5 = 0.995
\end{aligned}$$

and finally, to a_2:

$$\begin{aligned}
&\textbf{symbol} = (a_2),\\
&l^{(4)} = 0.990 + (0.995 - 0.990)0.990 = 0.99250\\
&u^{(4)} = 0.995 + (0.995 - 0.990)0.995 = 0.99450
\end{aligned}$$

Therefore, the resulting tag for the sequence (a_3, a_3, a_1, a_2) is:

$$\overline{T}_X(a_3a_3a_1a_2) = \frac{0.99250 + 0.99450}{2} = 0.9935$$

Then we have the tag 0.9935. The decoding is the reproduce process of the encoding steps. Thus, first, as at encoding, we start with $l^{(0)} = 0$ and $u^{(0)} = 1$ and recovers the tag value which is 0.9935. After decoding the first element of the sequence x_1, upper and lower bound becomes:

$$\begin{aligned}
l^{(1)} &= 0 + (1 - 0)F_X(x_1 - 1) = F_X(x_1 - 1)\\
u^{(1)} &= 0 + (1 - 0)F_X(x_1) = F_X(x_1)
\end{aligned}$$

The value of x_1, from the alphabet, where its interval $l^{(1)}$ and $u^{(1)}$ contains tag is a_3, because its interval ranges $[0.9, 1.0)$. Now the process is repeated with $l^{(2)}$ and $u^{(2)}$:

$$\begin{aligned}
l^{(2)} &= 0.9 + (1.0 - 0.9)F_X(x_2 - 1) = 0.9 + 0.1F_X(x_2 - 1) \qquad (6.1)\\
u^{(2)} &= 0.9 + (1.0 - 0.9)F_X(x_2) = 0.9 + 0.1F_X(x_2)
\end{aligned}$$

In this case:

$$\begin{aligned}
0.9 + 0.1F_X(x_2 - 1) \leq 0.9935 &< 0.9 + 0.1F_X(x_2)\\
0.1F_X(x_2 - 1) \leq 0.0935 &< 0.1F_X(x_2)\\
F_X(x_2 - 1) \leq 0.9350 &< F_X(x_2)
\end{aligned}$$

Testing for each x_2 value, we verify that with a_3 we have the same interval $[0.9, 1.0)$, which contains satisfies the inequation 6.1. Substituting the interval values at 6.1 and 6.1:

$$\begin{aligned}
l^{(3)} &= 0.99 + (1 - 0.99)F_X(x_3 - 1) = 0.99 + 0.01F_X(x_3 - 1)\\
u^{(3)} &= 0.99 + (1 - 0.99)F_X(x_3) = 0.99 + 0.01F_X(x_3)
\end{aligned}$$

Solving the inequation:

$$\begin{aligned}
0.99 + 0.01F_X(x_3 - 1) \leq 0.9935 &< 0.99 + 0.01F_X(x_3)\\
0.01F_X(x_3 - 1) \leq 0.0035 &< 0.01F_X(x_3)\\
F_X(x_3 - 1) \leq 0.3500 &< F_X(x_3)
\end{aligned}$$

Table 6.2: Example with contexts - Symbols context probability table

	Symbol		
Previous	a_1	a_2	a_3
a_1	0.4	0.2	0.4
a_2	0.1	0.8	0.1
a_3	0.25	0.25	0.5

Then, testing for all possibilities, we verify that a_1, with interval $[0.0, 0.5)$, satisfies the inequation 6.1. Repeating the process, we obtain as x_4 the symnol a_2, decoding the entire sequence. Its important to note that we stop the process because we already know the data length, consequently being that an important information to be passed along with the tag.

6.2 CONTEXT ARITHMETIC CODING

Let's observe an example of arithmetic encoding using contexts. Suppose that the message (a_1, a_3, a_3) will be encoded, and the its probabilities given a context is given in 2.8. In this example we will have an additional rescaling case where if the interval is entirely confined in the interval $[0.25, 0.75)$, the rescale is $E_3 = 2 \times (x - 0.25)$. When a E_3 happens, we increment a counter N_{E_3}. When E_1 case happens, we send bit 0 plus N_{E_3} bits 1 and reset N_{E_3}. Otherwise, when E_2 case happens, we send bit 1 plus N_{E_3} bits 0 and reset N_{E_3}.

In the beginning we assume that all the symbols are equiprobable. Thus, for a_1:

$$\begin{aligned}
&\mathbf{x} = (a_1), \quad code = (0),\\
&l^{(1)} = 0 + (1-0)0.0 = 0\\
&u^{(1)} = 0 + (1-0)0.33 = 0.3333
\end{aligned}$$

The interval is contained in the 1st half. Then a E_1 remap is performed and we send the bit 0:

$$\begin{aligned}
&\mathbf{x} = (a_1), \quad code = (0),\\
&l^{(1)} = 2 \times 0 = 0\\
&u^{(1)} = 2 \times 0.33 = 0.6666
\end{aligned}$$

For a_3, now given that a_1 precedes the symbol($F_X(a_3|a_1)$):

$$\begin{aligned}
&\mathbf{x} = (a_1, a_3), \quad code = (0),\\
&l^{(2)} = 0 + (0.6666 - 0)0.6 = 0.4\\
&u^{(2)} = 0 + (0.6666 - 0)1 = 0.6666
\end{aligned}$$

Rescaling using E_3, we increment the N_{E_3}:

$$\begin{aligned}
&N_{E_3} = 1, \quad \mathbf{x} = (a_1, a_3), \quad \textit{code} = (0), \\
&l^{(2)} = 2 \times (0.4 - 0.25) = 0.3 \\
&u^{(2)} = 2 \times (0.6666 - 0.25) = 0.8332
\end{aligned}$$

For a_3, using the context $F_X(a_3|a_3)$:

$$\begin{aligned}
&N_{E_3} = 1, \quad \mathbf{x} = (a_1, a_3, a_3), \quad \textit{code} = (0), \\
&l^{(3)} = 0.3 + (0.8332 - 0.3)0.5 = 0.5666 \\
&u^{(3)} = 0.3 + (0.8332 - 0.3)1 = 0.8333
\end{aligned}$$

Applying E_2, we send a bit 1 followed by N_{E_3} 0's:

$$\begin{aligned}
&N_{E_3} = 0, \quad \mathbf{x} = (a_1, a_3, a_3), \quad \textit{code} = (010) \\
&l^{(2)} = 2 \times (0.5666 - 0.5) = 0.1332 \\
&u^{(2)} = 2 \times (0.8333 - 0.5) = 0.6666
\end{aligned}$$

Finishing the symbols encoding. Then, we send the tag status, a value residing on the last interval obtained. Most of the times, the value of tag status is $l^{(n)}$. But in the interval $[0.1332, 0.6665)$, the most convenient value is 0.5, in fixed point being 10..00, having as many 0 as the word length of the implementation used. In this case, sending bit 1 suffices. Then, the final code is (0101).

From this code, we can decode it similarly by mimic the encoding process and guiding by the code sent, removing the most significant bit for each remaping occurrence. Thus, decoding this original data, we initialize $u^{(0)}$ and $l^{(0)}$, and starts to decode the sequence $\textit{code} = (0101)$. In this process, we assume that the decoder knows: the probability context table, the first symbol probability is equiprobable, the code sequence length is 3, and the code. Initializing $l^{(0)=0}$ and $u^{(0)=1}$, the decoding process follows:

$$\begin{aligned}
&\textit{tag}_{\textit{bin}} = 0101, \quad \textit{tag} = 0.3125, \\
&t_* = 0.3125, \quad l^{(0)} = 0, \quad u^{(0)} = 1.
\end{aligned}$$

With $F_X(0) = 0$, $F_X(1) = 0.33$, $F_X(2) = 0.66$, $F_X(3) = 1$. Because $t_* = 0.3125$ is between $[0, 0.33)$, the first symbol is a_1. Thus:

$$\begin{aligned}
&\textit{tag}_{\textit{bin}} = 0101, \quad \textit{tag} = 0.3125, \quad t_* = 0.3125, \\
&l^{(1)} = 0 + (1 - 0)0 = 0 \\
&u^{(1)} = 0 + (1 - 0)0.33 = 0.33
\end{aligned}$$

Rescaling using E_1 and consuming one bit 0:

$$\begin{aligned}
&\textit{tag}_{\textit{bin}} = 101, \quad \textit{tag} = 0.625, \quad t_* = 0.3125, \\
&l^{(1)} = 2 \times 0 = 0 \\
&u^{(1)} = 2 \times 0.3333 = 0.6666
\end{aligned}$$

and:

$$t_* = \frac{tag - l^{(1)}}{u^{(1)} - l^{(1)}} = \frac{0.625 - 0}{0.6666 - 0} = 0.9375$$

We know that $t_* = 0.9375 \in [F_X(2|a_1), F_X(3|a_1)) = [0.6, 1)$, then the symbol decoded is a_3. The new interval is:

$$\begin{aligned} l^{(2)} &= 0 + (0.6666 - 0)0.6 = 0.4 \\ u^{(2)} &= 0 + (0.6666 - 0)1 = 0.6666 \end{aligned}$$

which is confined in $[0.25, 0.75)$, requiring a E_3 remapping:

$$\begin{aligned} &tag_{bin} = 101, \quad tag = 0.625, \quad t_* = 0.3125, \\ &l^{(2)} = 2 \times (0.4 - 0.25) = 0.3 \\ &u^{(2)} = 2 \times (0.6666 - 0.25) = 0.8332 \end{aligned}$$

It‘s important to note that as E_3 does not send any bit in the encoding, as well will not consume any bit on the decoding, but count the times it occurs. Thus, updating t_*:

$$t_* = \frac{tag - l^{(2)}}{u^{(2)} - l^{(2)}} = 0.6095$$

which is confined in $[F_X(2|a_3), F_X(3|a_3)) = [0.5, 1)$. Finally, the last symbol is a_3, decoding all the message. If we proceeded the interval updating we would obtain:

$$\begin{aligned} &tag_{bin} = 101, \quad tag = 0.625, \quad t_* = 0.3125, \\ &l^{(3)} = 0.3 + (0.8332 - 0.3)0.5 = 0.5666 \\ &u^{(3)} = 0.3 + (0.8332 - 0.3)1 = 0.8333 \end{aligned}$$

that resides in the upper half, remaping using E_2 to

$$\begin{aligned} &tag_{bin} = 1, \quad tag = 0.5, \quad t_* = 0.3125, \\ &l^{(3)} = 2 \times (0.5666 - 0.5) = 0.1332 \\ &u^{(3)} = 2 \times (0.8333 - 0.5) = 0.6666 \end{aligned}$$

It’s important to note that we remaped E_3 followed by a E_2. Because of that, we consumes 2 bits representing the E_3 remap effect. In result, we have tag as 0.5. A interesting fact is that the tag value resides on this last interval range, satisfying and confirming the tag value interval. Therefore, the decoding was performed correctly synchronized and consistent with the encoding.

6.3 CONTEXT ADAPTIVE BINARY ARITHMETIC CODING

Given a overview about the CABAC that is implemented on this work and after used on the geometry code binary bitstream, a example will clarify the algorithm mechanism. A data sequence is defined as (000011), we must encode this message having a word length m equals 6, and 1 bit of context, i.e., previous bit as 0 or 1. Thus we first initialize the bounds $l^{(0)}$ and $u^{(0)}$, as 000000 and 111111 respectively. Updating the bounds:

Context/Symbol	0	1	**Total**
0	1	1	2
1	1	1	2

$$\begin{aligned}
&message = (000011), \quad bit = 0, \\
&l^{(0)} = 000000 && l^{(0)} = 0 \\
&u^{(0)} = 111111 && u^{(0)} = 63 \\
&l^{(1)} = 0 + \left\lfloor \frac{(63-0+1)0}{2} \right\rfloor = 0 && l^{(1)} = 000000 \\
&u^{(1)} = 0 + \left\lfloor \frac{(63-0+1)1}{2} \right\rfloor - 1 = 31 && u^{(1)} = 011111 \\
&E_1 : l^{(1)} = 000000 \quad u^{(1)} = 111111 \quad code = (0)
\end{aligned}$$

Being the first one, the context can't be updated yet. Proceeding to 2nd bit:

$$\begin{aligned}
&message = (00011), \quad bit = 0, \\
&l^{(1)} = 000000 && l^{()} = 0 \\
&u^{(1)} = 111111 && u^{(1)} = 63 \\
&l^{(2)} = 0 + \left\lfloor \frac{(63-0+1)0}{2} \right\rfloor = 0 && l^{(2)} = 000000 \\
&u^{(2)} = 0 + \left\lfloor \frac{(63-0+1)1}{2} \right\rfloor - 1 = 31 && u^{(2)} = 011111 \\
&E_1 : l^{(2)} = 000000 \quad u^{(2)} = 111111 \quad code = (00)
\end{aligned}$$

From this bit, we can update the context, having a 0 precending 0. Thus, to 3rd bit:

Context/Symbol	0	1	**Total**
0	2	1	3
1	1	1	2

$$
\begin{array}{ll}
message = (0011), \quad bit = 0, & \\
l^{(2)} = 000000 & l^{(2)} = 0 \\
u^{(2)} = 111111 & u^{(0)} = 63 \\
l^{(3)} = 0 + \left\lfloor \dfrac{(63-0+1)0}{3} \right\rfloor = 0 & l^{(3)} = 000000 \\
u^{(3)} = 0 + \left\lfloor \dfrac{(63-0+1)2}{3} \right\rfloor - 1 = 41 & u^{(3)} = 101001 \\
code = (00) &
\end{array}
$$

4th bit:

Context/Symbol	0	1	**Total**
0	3	1	4
1	1	1	2

$$
\begin{array}{ll}
message = (011), \quad bit = 0, & \\
l^{(4)} = 0 + \left\lfloor \dfrac{(41-0+1)0}{4} \right\rfloor = 0 & l^{(4)} = 000000 \\
u^{(4)} = 0 + \left\lfloor \dfrac{(41-0+1)3}{4} \right\rfloor - 1 = 30 & u^{(4)} = 011110 \\
code = (000) &
\end{array}
$$

5th bit:

Context/Symbol	0	1	**Total**
0	4	1	5
1	1	1	2

$$
\begin{array}{ll}
message = (11), \quad bit = 1, & \\
l^{(5)} = 0 + \left\lfloor \dfrac{(61-0+1)4}{5} \right\rfloor = 49 & l^{(5)} = 110001 \\
u^{(5)} = 0 + \left\lfloor \dfrac{(61-0+1)5}{5} \right\rfloor - 1 = 61 & u^{(5)} = 111101 \\
2 \times E_2 : l^{(5)} = 000100 \quad u^{(5)} = 110111 \quad code = (00011) &
\end{array}
$$

Finally, the 6th bit:

Context/Symbol	0	1	**Total**
0	4	2	6
1	1	1	2

$$\begin{aligned}
&message = (1), \quad bit = 1, \\
&l^{(6)} = 4 + \left\lfloor \frac{(55-4+1)1}{2} \right\rfloor = 30 && l^{(6)} = 011010 \\
&u^{(6)} = 4 + \left\lfloor \frac{(55-4+1)2}{2} \right\rfloor - 1 = 55 && u^{(6)} = 110111 \\
&code = (00011)
\end{aligned}$$

where we have the tag value residing between $[30, 55]$. Thus, as tag status value we send 32, which is 100000. So the final tag value becomes $code = (00011100000)$.

Then with the code generated, the decoding process follows. Having a word of length 6, the tag initialize with the 6 first bits from the code. As the rescaling happens, the t value is shifted to the left, and proceed appending at LSB the MSB bits from the code. Here we assume that $F(-1) = Cum\_Count(-1) = 0$. Thus, being the decoding similar to reproduce the steps from the coder, we have to the first bit decoded:

Context/Symbol	0	1	**Total**
0	1	1	2
1	1	1	2

$$t = 000111 = 7 \quad code = 00000 \quad Cum\_Count = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{6.2}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.3}$$

$$t^* = \left\lfloor \frac{(7-0+1)\times 2 - 1}{63-0+1} \right\rfloor = 0 \tag{6.4}$$

Analysing the tag value in the Cum_Count interval:

$$F(-1) \le t^* < F(0) \tag{6.5}$$

Then, the 1st bit from the original message is 0. Therefore, the updating is according to this decoded bit:

$$l^{(1)} = 0 + \left\lfloor \frac{(63-0+1)Cum\_Count(-1)}{2} \right\rfloor = 0 \qquad l^{(1)} = 000000 \tag{6.6}$$

$$u^{(1)} = 0 + \left\lfloor \frac{(63-0+1)Cum\_Count(0)}{2} \right\rfloor - 1 = 31 \qquad u^{(1)} = 011111 \tag{6.7}$$

$$\tag{6.8}$$

$$E_1 : l^{(1)} = 000000 \quad u^{(1)} = 111111 \tag{6.9}$$

Proceeding:

Context/Symbol	0	1	**Total**
0	1	1	2
1	1	1	2

$$t = 001110 = 14 \quad code = 0000 \quad Cum\_Count = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{6.10}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.11}$$

$$t^* = \left\lfloor \frac{(14-0+1)\times 2 - 1}{63-0+1} \right\rfloor = 0 \tag{6.12}$$

$$F(-1) \leq t^* < F(0) \Rightarrow message = (00) \tag{6.13}$$

$$l^{(2)} = 0 + \left\lfloor \frac{(63-0+1)0}{2} \right\rfloor = 0 \qquad l^{(2)} = 000000 \tag{6.14}$$

$$u^{(2)} = 0 + \left\lfloor \frac{(63-0+1)1}{2} \right\rfloor - 1 = 31 \qquad u^{(2)} = 011111 \tag{6.15}$$

$$E_1 : l^{(1)} = 000000 \quad u^{(1)} = 111111 \tag{6.16}$$

Updating the context table, and proceeding the decoding, we have:

Context/Symbol	0	1	**Total**
0	2	1	3
1	1	1	2

$$t = 011100 = 28 \quad code = 000 \quad message = (00) \quad Cum\_Count = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \tag{6.17}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.18}$$

$$t^* = \left\lfloor \frac{(28 - 0 + 1) \times 3 - 1}{63 - 0 + 1} \right\rfloor = 1 \tag{6.19}$$

$$F(-1) \leq t^* < F(0) \Rightarrow message = (000) \tag{6.20}$$

$$l^{(3)} = 0 + \left\lfloor \frac{(63 - 0 + 1)0}{3} \right\rfloor = 0 \qquad l^{(3)} = 000000 \tag{6.21}$$

$$u^{(3)} = 0 + \left\lfloor \frac{(63 - 0 + 1)2}{3} \right\rfloor - 1 = 41 \qquad u^{(3)} = 101001 \tag{6.22}$$

$$\tag{6.23}$$

The same process is repeated. At the fourth step:

Context/Symbol	0	1	**Total**
0	3	1	4
1	1	1	2

$$t = 011100 = 28 \quad code = 000 \quad message = (000) \quad Cum\_Count = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \tag{6.24}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.25}$$

$$t^* = \left\lfloor \frac{(28 - 0 + 1) \times 4 - 1}{41 - 0 + 1} \right\rfloor = 2 \tag{6.26}$$

$$F(-1) \leq t^* < F(0) \Rightarrow message = (0000) \tag{6.27}$$

$$l^{(4)} = 0 + \left\lfloor \frac{(41 - 0 + 1)0}{4} \right\rfloor = 0 \qquad l^{(4)} = 000000 \tag{6.28}$$

$$u^{(4)} = 0 + \left\lfloor \frac{(41 - 0 + 1)3}{4} \right\rfloor - 1 = 30 \qquad u^{(4)} = 011110 \tag{6.29}$$

$$E_1 : l^{(4)} = 000000 \quad u^{(4)} = 111101 \tag{6.30}$$

Fifth bit to decode:

Context/Symbol	0	1	**Total**
0	4	1	5
1	1	1	2

$$t = 111000 = 56 \quad code = 00 \quad message = (0000) \quad Cum\_Count = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \tag{6.31}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.32}$$

$$t^* = \left\lfloor \frac{(56 - 0 + 1) \times 5 - 1}{61 - 0 + 1} \right\rfloor = 4 \tag{6.33}$$

$$F(0) \leq t^* < F(1) \Rightarrow message = (00001) \tag{6.34}$$

$$l^{(5)} = 0 + \left\lfloor \frac{(61 - 0 + 1)4}{5} \right\rfloor = 49 \qquad l^{(5)} = 110001 \tag{6.35}$$

$$u^{(5)} = 0 + \left\lfloor \frac{(61 - 0 + 1)5}{5} \right\rfloor - 1 = 61 \qquad u^{(5)} = 111101 \tag{6.36}$$

$$2 \times E_1 : l^{(5)} = 000100 \quad u^{(5)} = 110111 \tag{6.37}$$

Finally, we know that the last decoded bit was:

Context/Symbol	0	1	**Total**
0	4	2	6
1	1	1	2

$$t = 100000 = 32 \quad code = \quad message = (00001) \quad Cum\_Count = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{6.38}$$

$$l^{(0)} = 0 \quad u^{(0)} = 63 \tag{6.39}$$

$$t^* = \left\lfloor \frac{(32 - 4 + 1) \times 2 - 1}{55 - 4 + 1} \right\rfloor = 1 \tag{6.40}$$

$$F(0) \leq t^* < F(1) \Rightarrow message = (000011) \tag{6.41}$$

$$l^{(6)} = 4 + \left\lfloor \frac{(55 - 4 + 1)1}{2} \right\rfloor = 30 \qquad l^{(6)} = 011110 \tag{6.42}$$

$$u^{(6)} = 4 + \left\lfloor \frac{(55 - 4 + 1)2}{2} \right\rfloor - 1 = 55 \qquad u^{(6)} = 110111 \tag{6.43}$$

$$\tag{6.44}$$

Which recovers our original message (000011). It's important to note that through the decoding process, the decode should happen before the Adaptive Context update step, otherwise its context table will not be the same as the encoder at each step. With this example and explanation, we can know understand it's application on S3D.

References

[1] L. Cui, R. Mekuria, M. Preda, and E. S. Jang, “Point-cloud compression: Moving picture experts group’s new standard in 2020,” *IEEE Consumer Electronics Magazine*, vol. 8, no. 4, pp. 17–21, 2019.

[2] R. Mekuria, K. Blom, and P. Cesar, “Design, implementation, and evaluation of a point cloud codec for tele-immersive video,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 4, pp. 828–842, 2017.

[3] O. Devillers and P.-M. Gandoin, “Geometric compression for interactive transmission,” in *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*, 2000, pp. 319–326.

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

[5] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler, and M. Pollefeys, “SEMANTIC3D.NET: A new large-scale point cloud classification benchmark,” in *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-1-W1, 2017, pp. 91–98.

[6] Y. Huang, J. Peng, C.-C. J. Kuo, and M. Gopi, “A generic scheme for progressive point cloud coding,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 2, pp. 440–453, 2008.

[7] 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. Goslar, DEU: Eurographics Association, 2006, p. 111–121.

[8] R. Rosário and E. Peixoto, “Intra-frame compression of point cloud geometry using boolean decomposition,” in *2019 IEEE Visual Communications and Image Processing (VCIP)*, 2019, pp. 1–4.

[9] E. Peixoto, “Intra-frame compression of point cloud geometry using dyadic decomposition,” *IEEE Signal Processing Letters*, vol. 27, pp. 246–250, 2020.

[10] D. R. Freitas, E. Peixoto, R. L. de Queiroz, and E. Medeiros, “Lossy point cloud geometry compression via dyadic decomposition,” in *2020 IEEE International Conference on Image Processing (ICIP)*, 2020, pp. 2731–2735.

[11] E. Peixoto, E. Medeiros, and E. Ramalho, “Silhouette 4d: An inter-frame lossless geometry coder of dynamic voxelized point clouds,” in *2020 IEEE International Conference on Image Processing (ICIP)*, 2020, pp. 2691–2695.

[12] “Filter design using matlab,” https://www.mathworks.com/discovery/filter-design.html?s_tid=srchtitle, accessed: 2021-08-27.

[13] “Transforms,” https://www.mathworks.com/help/signal/transforms.html?s_tid=srchtitle_transforms_9, accessed: 2021-08-27.

[14] “Memory management and cleanup,” https://www.mathworks.com/help/compiler_sdk/cxx/about-memory-management-and-cleanup.html, accessed: 2021-08-27.

[15] C. Loop, C. Zhang, and Z. Zhang, “Real-time high-resolution sparse voxelization with application to image-based modeling,” in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–79. [Online]. Available: https://doi.org/10.1145/2492045.2492053

[16] “Jpeg pleno database: Microsoft voxelized upper bodies - a voxelized point cloud dataset,” http://plenodb.jpeg.org/pc/microsoft, accessed: 2021-08-29.

[17] “Ply - polygon file format,” http://paulbourke.net/dataformats/ply/, accessed: 2021-08-29.

[18] K. Sayood, *Introduction to Data Compression.* Department of Electrical Engineering, University Nebraska-Lincoln, Lincoln, Nebraska: ACADEMIC PRESS, 2003.

[19] M. Krivokuća, P. A. Chou, and M. Koroteev, “A volumetric approach to point cloud compression–part ii: Geometry compression,” *IEEE Transactions on Image Processing*, vol. 29, pp. 2217–2229, 2020.

[20] “Google c++ style guide,” https://google.github.io/styleguide/cppguide.html, accessed: 2021-09-23.

[21] “Google test user’s guide,” https://google.github.io/googletest/, accessed: 2021-09-23.

[22] “Documenting the code,” https://www.doxygen.nl/manual/docblocks.html, accessed: 2021-09-28.