Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lossless intra compression of point cloud geometry based on S3D coder using threads and block decomposition (Compressão de geometria de nuvem de pontos intra sem perdas baseada no codificador S3D com uso de threads e decomposição em bloco)

Otho T. Komatsu

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Eduardo Peixoto Fernandes da Silva

Brasília
2024

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lossless intra compression of point cloud geometry based on S3D coder using threads and block decomposition (Compressão de geometria de nuvem de pontos intra sem perdas baseada no codificador S3D com uso de threads e decomposição em bloco)

Otho T. Komatsu

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Eduardo Peixoto Fernandes da Silva (Orientador)
Universidade de Brasília

Prof. Dr. Pedro Garcia Freitas
CIC/UnB

Prof. Dr. Luciano Volcan Agostini
UFPel

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Programa de Pós-graduação em Informática

Brasília, 19 de dezembro de 2024

Dedicatória

Agradeço a Deus pelo presente trabalho. Agradeço aos meus pais pelo cuidado e apoio no decorrer de todo o meu projeto, que sem isso não conseguiria chegar à metade do que progredi. E agradeço à Vitória por sempre estar presente me dando apoio e carinho no que precisasse, e estar ao meu lado acompanhando o progresso do projeto praticamente desde o início.

Agradecimentos

Gostaria de deixar agradecimentos especialmente à paciência e zelo do meu orientador Peixoto e Edil em me guiarem e darem as sugestões necessárias para o desenvolvimento do presente trabalho. Agradeço também à banca examinadora da minha qualificação de mestrado Luciano Agostini e Pedro Garcia pelos seus feedbacks e contribuições no meu trabalho, que levaram a dar mais riqueza e capricho nos resultados. Por fim, agradeço às instalações do laboratório GPDS para a coleta de resultados dos codificadores do projeto, e especialmente ao William, por estar presente frequentemente lá e me dar assistência quando fosse necessário.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Nas últimas décadas, o rápido avanço da tecnologia digital e da internet aumentou dramaticamente a demanda por dados. Esse crescimento nos requisitos de dados supera continuamente as capacidades de processamento do hardware e dispositivos atuais. Contra esse pano de fundo, o campo da compressão de sinais desempenha um papel fundamental na era digital de hoje. Nuvens de pontos, que representam cenas 3D por meio de um conjunto de pontos dentro de um espaço definido, emergiram como um formato significativo de nova mídia. As demandas de armazenamento de nuvens de pontos dinâmicas, que capturam cenas 3D com movimento ao longo do tempo, semelhantes a vídeos, são particularmente desafiadoras, sublinhando a necessidade crítica de pesquisa em compressão de nuvem de pontos. Enquanto técnicas tradicionais de compressão estabeleceram a base para o desenvolvimento de novos codecs, os codificadores mais populares se baseam na representação de geometria octree. No entanto, duas características cruciais que faltam nos codecs recentes são a concorrência e a codificação em tempo real de aquisição. Este trabalho introduz melhorias no codec Silhouette 3D (S3D), especificamente projetado para abordar essas lacunas. Apresentamos duas variações de multi-threading, S3D-Subtree (S3D-S) e S3D-Subtree+Toptree (S3D-ST), que possibilitam o processamento concorrente, uma característica distintiva entre os codecs de geometria de nuvem de pontos de última geração. Adicionalmente, propomos os codecs S3D Block Mode (S3D-BM) e S3D Inverted Mode (S3D-IM), oferecendo alternativas mais simples e diretas à decomposição diádica do S3D para contextos de aquisição em tempo real. Ao abordar a concorrência e a codificação em tempo real, este trabalho avança significativamente o campo da compressão de point clouds, possibilitando aplicações mais eficientes e práticas de representações de cenas 3D.

Palavras-chave: Nuvem de pontos, Compressão de sinais, Compressão de geometria, compressão intraframe;

Abstract

In the last few decades, the rapid advancement of digital technology and the internet has dramatically increased the demand for data. This growth in data requirements continuously outpaces the processing capabilities of current hardware and devices. In this scenario, the field of signal compression plays a pivotal role in today's digital era. Point clouds, which represent 3D scenes through a set of points within a defined space, have emerged as a significant new media format. The storage demands of dynamic point clouds, which capture 3D scenes with motion over time, akin to videos, are particularly challenging, underscoring the critical need for research in point cloud compression. While traditional compression techniques have laid the foundation for new codec development, the most popular codecs are based in the octree geometry representation. However, two crucial features lacking in recent codecs are concurrency and real-time acquisition encoding. This work introduces enhancements to Silhouette 3D (S3D) codec, specifically designed to address these gaps. We present two multi-threading variations, S3D-Subtree (S3D-S) and S3D-Subtree+Toptree (S3D-ST), which enable concurrent processing, a distinctive feature among state-of-the-art point cloud geometry codecs. Additionally, we propose the S3D Block Mode (S3D-BM) and S3D Inverted Mode (S3D-IM) codecs, offering simpler and more direct alternatives to the dyadic decomposition of S3D for real-time acquisition contexts. By addressing concurrency and real-time encoding, this work significantly advances the field of point cloud compression, enabling more efficient and practical applications of 3D scene representations.

Keywords: Point Clouds, Signal Compression, Geometry Compression, intraframe encoding

Sumário

1 Introduction **1**

1.1 Context . . . 1
1.2 Motivation . . . 5
1.3 Problem . . . 7
1.3.1 Multi-thread S3D . . . 8
1.3.2 Block Mode . . . 8
1.4 Purpose . . . 9

2 Literature Review **12**

2.1 Point Clouds . . . 12
2.2 Signal compression . . . 14
2.2.1 Geometry compression . . . 16
2.3 CABAC . . . 17
2.3.1 Arithmetic Coding . . . 18
2.3.2 Context Arithmetic Coding . . . 21
2.3.3 Context Adaptive Arithmetic Coding . . . 21
2.3.4 Context Adaptive Binary Arithmetic Coding . . . 22
2.4 Silhouette 3D . . . 24
2.4.1 Silhouette . . . 25
2.4.2 Dyadic Decomposition . . . 25
2.4.3 Single Mode . . . 31
2.4.4 Best choice . . . 33
2.5 Threads . . . 35
2.5.1 Multi-threading . . . 35

3 Proposed Algorithms **39**

3.1 Multi-threaded S3D . . . 39
3.1.1 S3D-subtree(S3D-S) . . . 42
3.1.2 S3D-subtree + Top-tree (S3D-ST) . . . 43

3.2 S3D Block Mode 44
3.2.1 Standard (S3D-BM) 46
3.2.2 Inverted Mode (S3D-IM) 47
3.3 Unified header 48
3.4 Arithmetic Coding Contexts 50

4 Results and analysis **52**
4.1 2D and 3D contexts setting 53
4.2 Standard S3D C++ algorithm results 54
4.3 S3D-S and S3D-ST results 56
4.3.1 Running time 57
4.3.2 Rate 59
4.4 S3D-Block Mode (S3D-BM) and S3D-Inverted Mode (S3D-IM) results . . . 62
4.5 S3D algorithms benchmark 64

5 Conclusions **71**

Referências **74**

Lista de Figuras

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

2.1 The octree geometry compression illustration 17
2.2 Interval from arithmetic coding tag generating 20
2.3 Point cloud sliced and its respective silhouettes projected 26
2.4 Ricardo9 point cloud projections 27
2.5 Binary tree from point cloud recursive slices. 28
2.6 A silhouette decomposition illustration 29
2.7 A silhouette decomposition example with 3 levels 30
2.8 Contexts used to encode a pixel 31
2.9 The encoding process of the binary tree node using contexts. 32
2.10 Slices images merged 33
2.11 Representation of the single mode operations - this figure illustrates a point cloud of length L, resulting in L slices during the single mode operation. . 34
2.12 State diagram of mutex variable operations 38

3.1 Representation of the decomposition tree from the dyadic decomposition. . 40
3.2 Triad of nodes from each node half-decomposition. Each one has a father, left child node and right child node. 41
3.3 Triad of nodes from each node half-decomposition. Each one has a father, left child node and right child node. 41
3.4 The decomposition tree divided in each 4 subtrees beginning from the 3rd level. 42
3.5 Decomposition tree of the S3D-S algorithm - highlighting the root node for each subtree. 43

3.6 Decomposition tree of the S3D-ST algorithm - highlighting the root node for each subtree. Here, we can observe that the encoding of the subtrees and the root node are in different threads 45
3.7 Decomposition tree of the S3D-standard-block-mode algorithm - in the case of this example, $k = 2$ and $m = 4$ 46
3.8 Decomposition tree of the S3D-inverted-mode algorithm - in the case of this example, $k = 2$ and $m = 4$ 47
3.9 The general decomposition tree of the S3D block mode algorithm. Here are clear the basic structure of the algorithm, and how it can branch in the different algorithms in figures 3.7 and 3.8, changing the tree traversing (top-down or bottom-up). 48
3.10 Unified header structure illustration, relating fields and its bit length. . . . 49
3.11 3D pixels contexts sequence for the previous and new version of the S3D. . 51

4.1 S3D standard contexts combination scatter plot on rate x time distribution, bits per voxel x seconds 54
4.2 S3D standard contexts combination scatter plot for 3D contexts equal to 9, bits per occupied voxel vs seconds 55
4.3 Previous version of S3D utilizes pixel contexts, with five 2D contexts and nine 3D contexts. In the case of encoding only with 2D contexts, the selected contexts are represented in (a). When 3D contexts are used, the selected contexts are represented in (b). 57
4.4 New version of S3D utilizes pixel contexts, with eight 2D contexts and nine 3D contexts. In the case of encoding only with 2D contexts, the selected contexts are represented in (a). When 3D contexts are used, the selected contexts are represented in (b). 58
4.5 Average Running Time for the first 12 frames from each dataset over different amount of threads for S3D-S and S3D-ST. 58
4.6 Average Rate for the first 12 frames from each dataset over different amount of threads for S3D-S and S3D-ST. 59
4.7 Rate and running time plot over different amount of threads for S3D-S. . . 62
4.8 Rate and running time plot over different amount of threads for S3D-ST. . 63
4.9 Block mode performance for *upper bodies* datasets across various k values. 65
4.10 Block mode performance for *full bodies* datasets across various k values. . . 65
4.11 Inverted mode performance for each point cloud from *upper bodies* dataset across various k values. 66
4.12 Inverted mode performance for each point cloud from *full bodies* dataset across various k values. 66

Lista de Tabelas

2.1 Initial Context Table 22
2.2 Context Table after Encoding a_1 22
2.3 Context Table after Encoding All Symbols 22

4.1 Results of rate (bits per occupied voxel) and time (seconds) from standard S3D encoding based on different amount of 2D and 3D contexts, sorted in descending order of mean_rate 56
4.2 S3D standard comparison with state-of-the-art codecs for the first 100 frames average shown in bits per occupied voxel (bpov). 57
4.3 Rate results using the first 12 frames average of each sequence. All rates are in bits per occupied voxel (bpov). 61
4.4 Running time results using the first 12 frames average of each sequence. All running time are in seconds (s). 61
4.5 Comparison of Block mode with k = 2 with state-of-the-art codecs for the first 100 frames from *upper bodies* and *full bodies* datasets, shown in bits per occupied voxel (bpov). 67
4.6 Comparison between block mode and its inverted mode for the first 100 frames from from *upper bodies* and *full bodies* datasets, with k equals 2 . . 68
4.7 Rate benchmark of the work's proposed algorithms, average of first 4 frames from each dataset, sampled 3 times, shown in bits per occupied voxel (bpov). 69
4.8 Running time benchmark of the work's proposed algorithms, average of first 4 frames from each dataset, sampled 3 times, shown in seconds (s). . . 70
4.9 Product of rate and time benchmark of the proposed algorithms from the work, average of first 4 frames from each dataset, sampled 3 times, shown in bits per occupied voxel per second (*bpov.s*) 70

Capítulo 1

Introduction

In this chapter, we will first explore the significance and applications of point clouds in both industrial and academic settings. Next, we will provide a brief introduction to the primary algorithm upon which this work is based, followed by an analysis of its notable weaknesses and strengths. By examining these aspects, we aim to propose new variations that mitigate the identified weaknesses and enhance the strengths, thereby improving the algorithm. This constitutes the core objective of our research.

1.1 Context

Technological advancements in scanning and detecting three-dimensional (3D) objects using specialized cameras represent a significant milestone and a recent trend in the industry. With the advent of more consumer-accessible devices, the application of 3D models has expanded beyond research to include industry and consumer markets. Examples include virtual reality (VR) broadcasting, 3D videos, robotics, autonomous navigation using large-scale dynamic 3D maps, geographic information system (GIS) applications, tele-immersive applications [1], work-related applications [2], animations, gaming, and scientific visualization [1].

A particularly interesting application is the creation of free viewpoint videos, where the movements of active individuals and objects are captured and transmitted in real-time to remote locations, allowing them to be viewed as 3D motion data from multiple positions and angles. This rendering of 3D objects to remote locations facilitates collaboration as if all parties were physically together. Such captures are made possible by arrangements of multiple cameras (both infrared and standard), with real-time capturing and rendering enabled by powerful graphics processing units (GPUs). The rendered images can then be viewed through specialized rendering glasses [3]. The surge in research and growing market demand for these technologies, fueled by recent innovations, has led to the development of

alternative methods for capturing and representing the data of these geometric volumes, each aiming to address limitations of standard methods and optimize data representation alongside hardware advancements.

With recent strides in 3D sensing and high-performance computing, point clouds have attracted increased attention [1]. Mobile devices such as Apple's iPhone X and Sony's Xperia XZ1 now support point-cloud representations of up to several hundred thousand points. However, as point cloud technology advances, allowing for the detailed representation of complex objects, a significant challenge emerges: raw point cloud data require a vast amount of storage space. This issue becomes particularly acute with the demand for larger, high-quality models, which store extensive data including geometry (i.e., more points), color, normals, reflectance, and other attributes [1]. This demand for larger representations and the consequent storage challenges are illustrated in figures 1.1 and 1.2, showcasing detailed examples of point cloud data [4]. Given the need for many systems and applications to utilize these large, high-quality point clouds for storage, transmission, processing, and rendering—especially in real-time applications—the efficient compression of this data is paramount. Consequently, the development of compression technologies for point clouds has become a field of intense research activity [1].

A primary concern in the field of 3D model compression is decoding efficiency, which ensures that end users experience swift reconstruction times for compressed models. This efficiency necessitates straightforward decoder implementations. Additionally, the codec's memory usage is a critical factor, as it directly influences the compression ratio [5]. The technique for coding 3D models has been under investigation for over a decade. While 3D mesh compression once dominated the discourse on graphics compression, the advent of point cloud technology has shifted focus towards the compression of point cloud data, making it a fervently researched area in recent times. In Queiroz et al. [3], an algorithm was proposed for the compression of the colors of 3D point clouds, based on the region-adaptive hierarchical transform, RAHT, and arithmetic coding driven by Laplacian distribution models. Besides, in Huang et al. [5] is proposed a generic point cloud encoder capable of compressing different attributes of a point cloud, such as position, color, normal with arbitrary topology. In this work is proposed the iterative octree cell subdivision model over its geometry. In 2014, MPEG initiated an exploratory activity on Point Cloud Compression (PCC). Reflecting the growing interest in point-cloud-based applications within the industry, MPEG issued a call for proposals (CfP) for PCC in 2/017 [6]. This led to the evaluation of 13 technical proposals in October 2017, resulting in the selection of three different technologies as test models for three distinct content categories [1]. Since then, the academic and industrial sectors have witnessed an influx of research papers and method proposals for point cloud compression, underscoring the

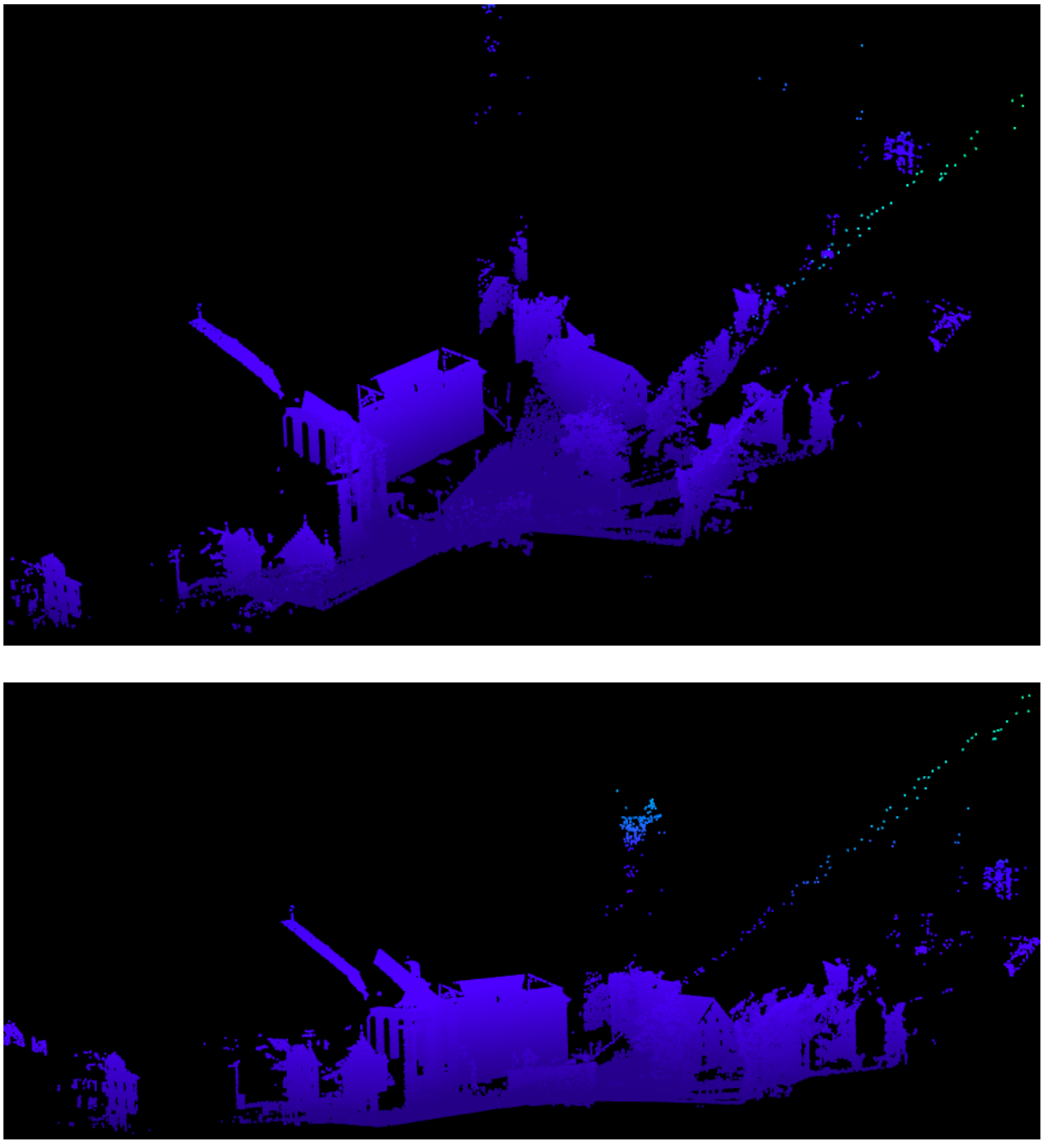


Figura 1.1: Large point cloud example 1 - St. Gallen Cathedral point cloud 3d model (32.342.450 points).

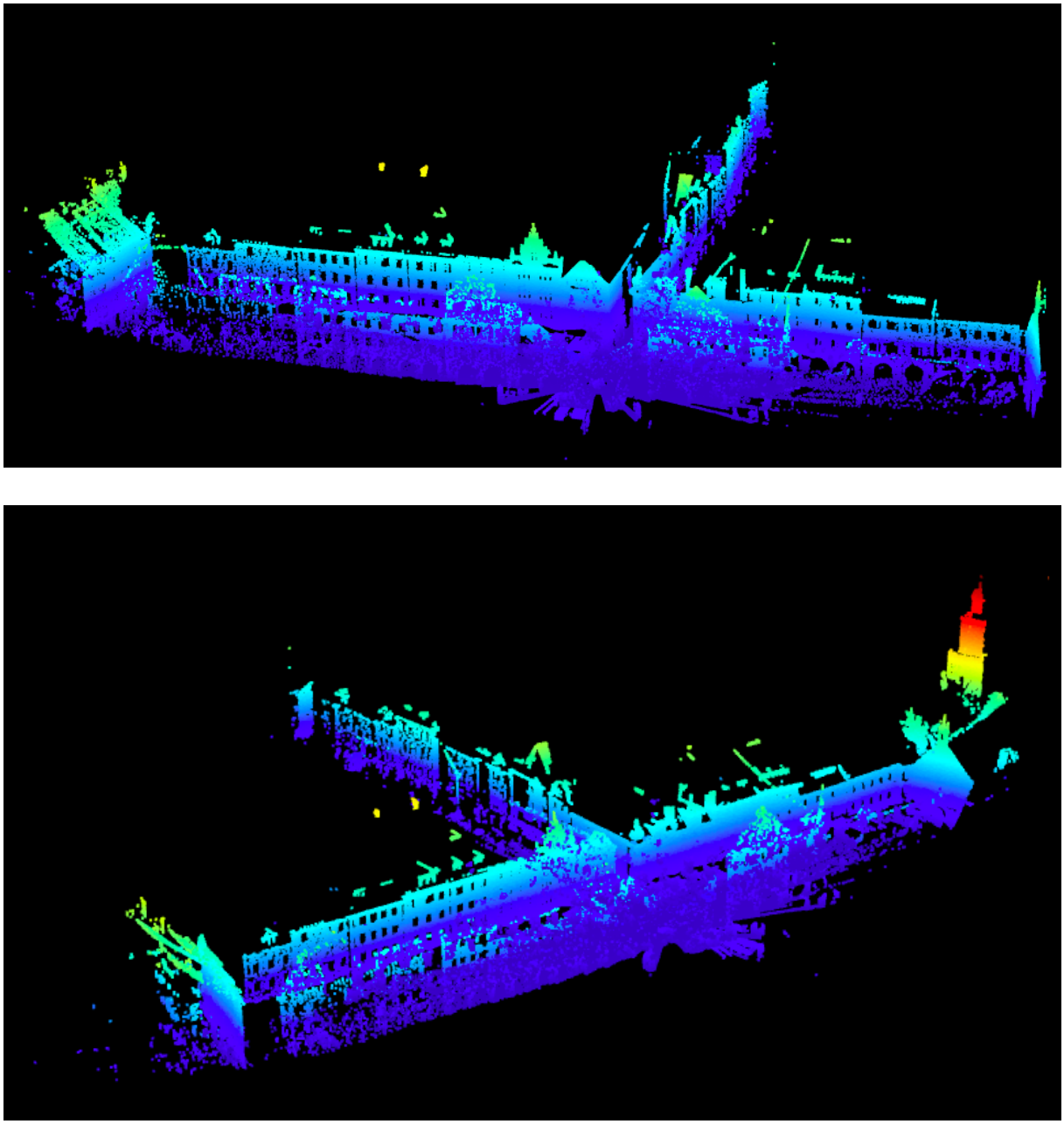


Figura 1.2: Large point cloud example 2 - Marketplace Feldkirch point cloud 3d model (22.760.334 points)

growing importance of this research area.

1.2 Motivation

Among the researches and papers published on point cloud compression techniques, a notable contribution came in 2019, when an intra-frame geometry compression method was presented at the IEEE VCIP. Their method leveraged JBIG and a novel approach based on Boolean Decomposition [7]. Building on this foundation, a subsequent technique, inspired by the Boolean Decomposition, was introduced in 2020 [8]. This method, titled "Lossless Intra-Frame Compression of Point Cloud Geometry Using Dyadic Decomposition,"employs a strategy of recursively splitting the original point cloud into halves, creating child point clouds. For each child, a silhouette of the occupied volume is generated along a specific axis. This process, akin to the previously mentioned Boolean Decomposition, is illustrated in figure 1.3 [9], but is distinguished by its reliance on Silhouette Decomposition. The encoding of silhouette decompositions utilizes a Context Adaptive Binary Arithmetic Coder, alongside a single mode in the dyadic decomposition process, leading to the development of the Silhouette-3D (S3D) technique. This methodology will be elaborated upon in the literature review section.

When comparing compression rates, specifically in terms of bits per occupied voxel (bpov), on two public databases, S3D surpasses all tested state-of-the-art intra coders, including the MPEG G-PCC TMC13 v7.0, and even a recent state-of-the-art inter coder, Parent Node Inheritance plus Super-Resolution P (Full)[8]. Additionally, a lossy approach employing downsampling and the omission of consecutive slices outperformed TMC13 v7.0's Trisoup and Octree for bitrates above 0.3 bpov in point-to-point error (C2C) metrics[9].

Advancing the S3D concept to accommodate dynamic point clouds, which incorporate a temporal dimension to represent motion in point cloud videos, a new technique was introduced. This inter-frame lossless geometry coder for dynamic voxelized point clouds, published in the same year, builds upon the S3D algorithm by incorporating a fourth pixel context during the encoding process. This addition enables the use of pixel data from another frame to aid in encoding, leading to the development of the Silhouette-4D (S4D) method. This method, in addition to considering the pixel contexts from the current silhouette, sibling silhouette, and parent, also factors in pixels from previous frames, as depicted in figure 1.4. The S4D demonstrates superior lossless compression performance compared to well-known techniques on the JPEG Pleno Database [10].

The novel methodologies and impressive performance outcomes based on the Dyadic Decomposition approach and the foundational S3D technique have opened new avenues for

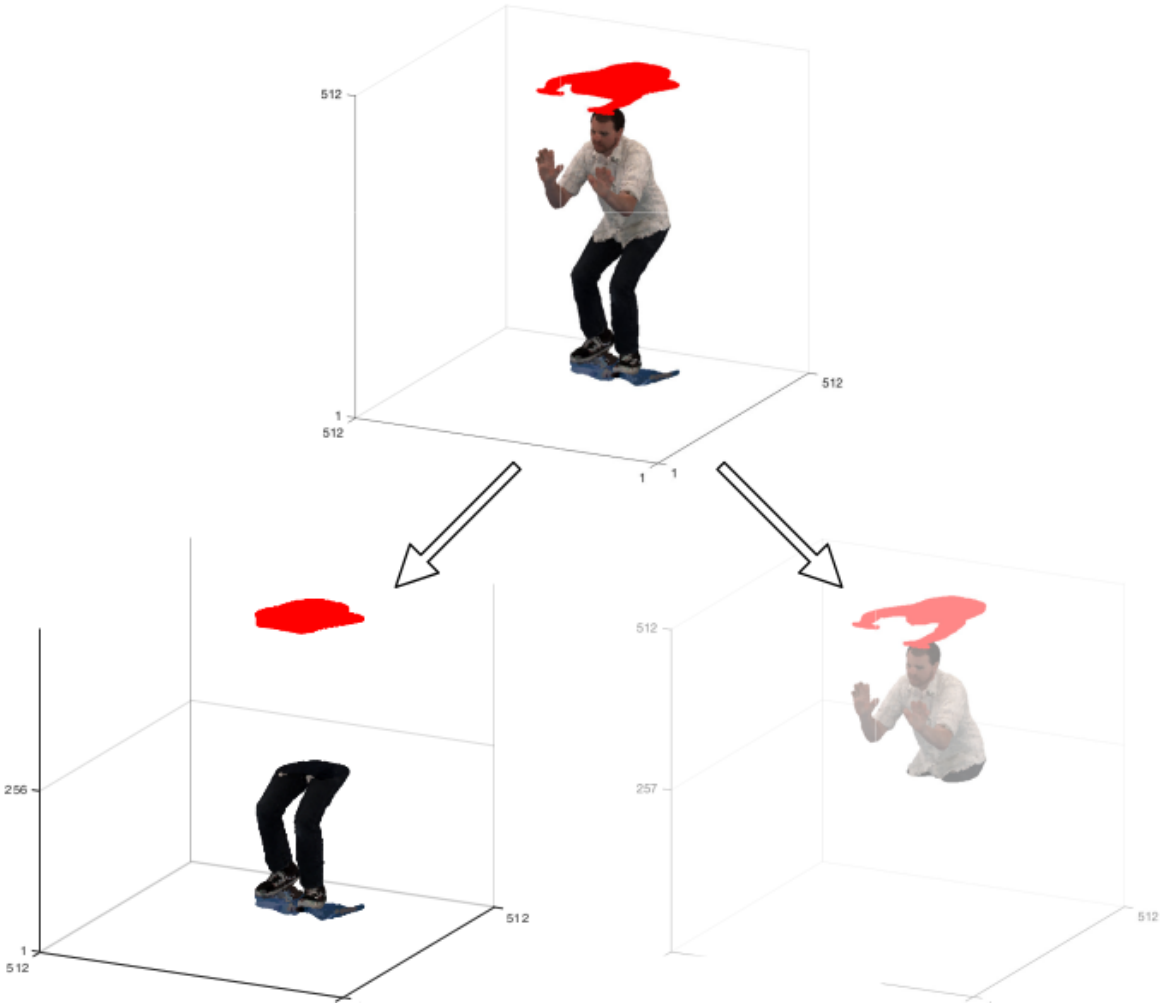


Figura 1.3: Dyadic decomposition on a single point cloud ilustrated.

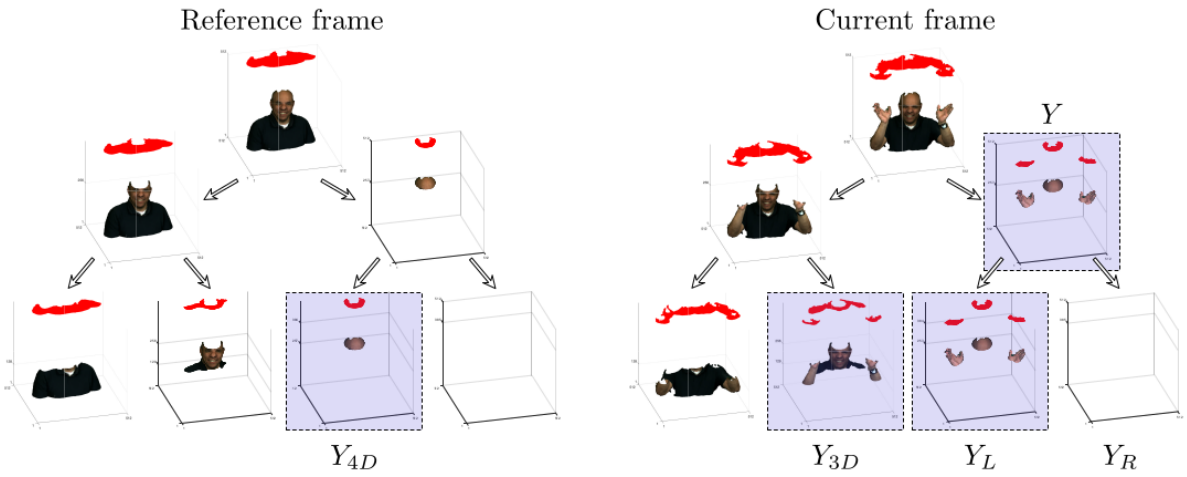


Figura 1.4: S4D decomposition process and context gathering.

further research and development of this algorithm. Efforts have been made to optimize this technique and delve deeper into the intricacies of the algorithm, aiming to produce refined versions that offer improved results and performance. The S3D algorithm and the S4D algorithm were originally implemented on the Matlab platform. While Matlab provides syntactic ease and high-performance operations for matrix manipulations, it is not typically chosen for codec development. As a non-compiled, interpreted language, codecs developed in Matlab suffer from slower execution times. Additionally, Matlab's garbage collection and memory management mechanisms limit developers' control over structure allocation. Within the context of point cloud geometry encoding, this restriction hampers the true potential of the algorithm and introduces time overhead due to garbage collection of Matlab and built-in memory management tools.

Given these limitations, a shift to a more conventional programming language for codec development was proposed to address the shortcomings of the Matlab implementation: a C++ version of the S3D algorithm. Consequently, in 2021, a project was launched to migrate the S3D algorithm from Matlab to C++. This C++ iteration adheres to the original strategy outlined in [8]. Finishing this new C++ implementation, and benchmarking the codec, was the first contribution of this work.

1.3 Problem

The definitive C++ implementation of the S3D algorithm has paved the way for an in-depth exploration of its capabilities and various adaptations. Since the initial version was implemented, different approaches inspired by the original S3D proposal have been developed, as in Alves's work [11], which proposes a parallelization of the S3D algorithm.

One of the main distinguishing features of our geometry decomposition approach, compared to others, is the decomposition tree derived from dyadic decomposition and single mode. As detailed further in Section 2, this tree can be interpreted as comprising multiple independent subtrees. This characteristic, combined with the scarcity of multi-threaded geometry decomposition algorithms, makes the exploration of a multi-threaded S3D algorithm particularly compelling.

Another noteworthy aspect of the decomposition tree is the computational cost of its construction. At each node, dyadic decomposition and single mode must be evaluated to determine the optimal solution. Given that tree height is directly proportional to the resolution of the point cloud, processing very large point clouds can be significantly time-consuming. In light of this, the S3D algorithm requires a more streamlined and straightforward approach to achieve comparable results at a similar cost.

In this work, we aim to explore four distinct algorithms: two utilize a multi-threaded approach (**Subtree** and **Subtree-Top**), and two employ a predefined decomposition strategy of the point cloud, referred to as **Block Mode**. We will first delve into the multi-threaded strategies before addressing the Block Mode approach.

1.3.1 Multi-thread S3D

One of the more recent projects building upon this C++ implementation, which seeks to explore new methods and explore the strengths of the algorithm, was introduced in a 2022 undergraduate thesis by Alves [11]. This thesis examines the feasibility of adapting the algorithm to utilize threads in segments where concurrency and processing can be independently managed. It proposes three algorithms: I, C, and S.

This approach, leveraging thread-based encoding, showcases promising results for optimizing time costs and broadening applicability across various domains, thus marking the thread-based S3D algorithm as a noteworthy continuation in development. Moreover, it is important to note that the threaded S3D algorithm stems from the initial version of the C++ project, focusing solely on dyadic decomposition. Consequently, Alves's work did not fully explore the original S3D algorithm through a threaded lens. Additionally, the methodology for decomposing the point cloud—termed the decomposition tree—is initially loop-based. The proposed new approach shifts towards a recursion-based model, where the decomposition process is driven by a main function that recursively calls itself, ceasing only upon reaching the leaf nodes, which represent silhouettes of 1-length blocks within the point cloud. The recursion implementation was adopted in this first version considering the simplicity and similarity with the concept. Compared to an alternative approach, iteration for instance, it involves a more complex abstraction and difficulty in its implementation.

Therefore, this initiative advocates for the comprehensive development of the complete S3D algorithm, inspired by Alves's threading concepts as introduced in algorithms I, C, and S. The resultant algorithms, **S3D Subtree (S3D-S)** and **S3D Subtree-Top (S3D-ST)**, aim to further refine and extend the capabilities of the S3D framework [12].

1.3.2 Block Mode

In the course of studying the application of threads to the S3D algorithm, various approaches to the decomposition of the recursion tree were considered and tested, targeting potential improvements in different aspects of the algorithm, notably its computational time cost. A significant portion of time and computational complexity in the original S3D proposal was attributed to two main factors: the need to test the compression rate

between Dyadic Decomposition (DD) and Single Mode (SM) for each node in the recursion tree, and the rigid approach to point cloud decomposition, which always splits it into halves. For example, a point cloud represented with 9 bits would generate a recursion tree of 1023 nodes. With each node offering two encoding possibilities—DD and SM—the process could potentially evaluate 2046 different encoding, selecting only the most efficient outcome.

The viability of this trade-off between performance and time cost largely depends on the context. For a pre-captured point cloud with all points already defined, where the immediacy of data availability is not critical, prioritizing performance despite increased time may be acceptable. Conversely, in scenarios where the point cloud is dynamically evolving with constantly added new points, the system cannot afford the luxury of extensive processing times to make all frames of point cloud data available. In such cases, agility becomes paramount, making time cost a far more critical consideration than compression performance.

Given these considerations, an approach that directly decomposes the point cloud into blocks of a predefined width and encodes each block through a more straightforward process, such as single mode, emerges as notably advantageous in terms of speed. For instance, since the encoding process slices the point cloud along a specific axis—processing these slices into blocks—this model is particularly well-suited for applications involving real-time acquisition, where points from a scene are progressively scanned to form the complete point cloud. This principle forms the foundation of the Block Mode method proposed in this work.

1.4 Purpose

In pursuit of a contemporary implementation of the S3D algorithm, the initial objective of this work is to develop the original S3D algorithm inclusive of the single mode option—addressing a gap left by the previous migration to C++. Moreover, this iteration introduces a comparison at each node within the decomposition tree between the compression rates of dyadic decomposition and single mode, selecting the more effective approach for each. The underlying assumption is that optimizing performance at the node level will inherently enhance overall encoding efficiency.

Completing the C++ implementation of the original S3D codec enables the exploration and evaluation of its performance when threading is applied across each subtree within the decomposition tree. This process unveils two distinct methodologies for integrating thread-based encoding:

1. **Subtree Method (S3D-S)**: This involves slicing the point cloud into halves until the number of subtrees aligns with the number of available threads. For example, with four threads, the tree decomposes to the third level, resulting in exactly four subtrees. Each subtree is then independently encoded by a dedicated thread, employing the S3D codec strategy.

2. **Subtree-Top Method (S3D-ST)**: Similar to the Subtree Method in initial decomposition, this strategy diverges by not encoding the root node of each subtree. To compensate, the top portion of the original recursion tree is encoded up to the level where the subtrees begin. This approach ensures that information about the unencoded root nodes is encapsulated within the final bitstream. Consequently, this method necessitates an additional thread for encoding the top part of the tree, increasing the total number of threads by one—for instance, five threads for four subtrees plus the top part. The advantage is that these root nodes are compressed more efficiently in terms of rate.

These methodologies necessitate the development of two threaded S3D codec variants. Previously, in the original S3D threaded based work, similar approaches were named as algorithms I and C. To distinguish from the original adaptations, we propose naming these new variants the S3D codec Subtree (S3D-S) and S3D codec Subtree-Top (S3D-ST).

Complementing the thread-based encoding methodologies, the Block Mode strategy introduces a streamlined, efficient algorithmic approach. This method facilitates the direct partitioning of the point cloud into a pre-determined number of slices, where each width of slice aligns with a power of two. This approach eliminates the necessity for iterative bisecting typically seen in dyadic decomposition, opting instead for a direct partition at a specific decomposition level.

A notable illustration of this strategy is the direct decomposition of a 9-bit point cloud into four blocks. Rather than sequentially halving the point cloud through to the third level of dyadic decomposition, this method immediately sections the point cloud into four equally sized blocks, each 128 pixels wide. Subsequently, each block undergoes encoding via the single mode approach, sidestepping the need to evaluate the most efficient compression strategy for each segment. Thus, by utilizing the single mode, instead of fully decomposing the point cloud partition in the dyadic decomposition tree, it directly decomposes the point cloud into unit-width slices that form the partition. This example underscores the unique advantage of the Block Mode approach: by avoiding the conventional dyadic decomposition in favor of immediate, level-specific partitioning, it significantly enhances the speed and simplicity of the encoding process.

This development leads to two procedural variants: a top-down and a bottom-up approach, differing only in the sequence of encoding. The top-down variant, or Block Mode,

begins with high-level nodes before addressing block nodes via single mode. Conversely, the bottom-up variant, dubbed Inverted Mode, starts with leaf node encoding, subsequently integrating these silhouettes into larger blocks. Originally, the Block mode proposal was based only on the concept of top-down traversal. However, the bottom-up approach, as will be explained in more detail in section 3, was thought to use its structure in specific situations. These situations are those in which the point cloud is not fully captured. Then, parts of the point cloud are captured, and then, merged into a full captured. Availing the Block mode decomposition in the bottom-up traversal, motivated the usage of the Inverted mode in these contexts. These methodologies and their/ implications will be delineated throughout this work.

This work builds on the Silhouette 3D codec [8], which employs the concept of dyadic decomposition for point cloud geometry. The first proposed algorithm explores the encoding of independent units over the subtrees created in the decomposition, facilitating multi-threading. This concept was previously explored in earlier works [11], though those studies were based on an earlier, non-definitive version of the S3D C++ implementation. The current work, therefore, proposes an updated version of the multi-threaded S3D algorithm, providing a deeper understanding of its capabilities in a multi-threaded context.

Additionally, a new approach called the S3D Block Mode is introduced, which employs a novel concept of block decomposition. This method is designed to address the computational cost associated with the original approach. The S3D Block Mode simplifies the process, making it particularly suited for applications that require time-efficient codecs. Furthermore, this mode allows partial encoding of the point cloud, focusing on slices of the captured object (e.g., ambient environments or moving objects). This feature makes it an excellent candidate for real-time acquisition applications.

The proposed S3D variants and their underlying concepts will be explored in detail, and their capabilities will be assessed. The success of these variants will be evaluated based on the results. However, before delving into the specific contributions of this work, it is essential to review fundamental concepts of point cloud compression and examine the Silhouette 3D codec in greater detail, as it forms the foundation of this research.

Capítulo 2

Literature Review

Given the increasing relevance of point clouds today and the aim of this work to contribute to point cloud geometry compression techniques, this section delve deeper into the study of point clouds and the primary algorithm upon which this research is based. Initially, this section examine the formal definition of a point cloud and the fundamental concepts underlying its geometry compression.

Subsequently, we will describe the arithmetic encoder, a crucial component of the S3D algorithm. We will also explore the main concepts associated with the S3D. Lastly, in light of the multi-threaded S3D variant proposed in this research, we will introduce elementary concepts of concurrent programming and multi-threading.

2.1 Point Clouds

A point cloud is defined as a set $\mathcal{V}$ of points [1]. The process known as *voxelization* maps the points of a point cloud onto a 3D discrete grid, with each cubic unit of space within this grid referred to as a *voxel*. In the context of this work, voxels serve as the 3D counterparts to 2D image pixels, representing index triples that denote a spatial location within a 3D grid of dimension $2^i \times 2^j \times 2^k$, where i, j, and k is a level within the voxel hierarchy $\mathcal{V}^k$. Associated with each voxel may be multiple attributes such as color and reflectance. Voxelized point clouds are typically captured by specialized cameras and undergo a voxelization process that assigns attribute values and geometric information. The quantity of voxels within a point cloud is denoted as $|\mathcal{V}^k|$, a measure proportional to the surface area of the object. The term *depth* refers to the maximum number of levels and is determined by the bit depth used for coordinate representation [13].

Point clouds are categorized as either static or dynamic, depending on the nature of their captured environment. Static point clouds represent unchanging data over time, devoid of motion or temporal variations. Conversely, dynamic point clouds incorporate

a temporal domain, capturing motion and changes across frames, which is essential for applications such as 3D video, autonomous navigation with dynamic 3D maps, and VR broadcasting [1]. To accommodate the diverse requirements of 3D applications, MPEG PCC classifies point clouds into three categories, each tailored to specific compression technologies and applications, as detailed in [1].

Polygon File Format (PLY) is the standard file representation for point clouds according to MPEG PCC standards [1]. In this format, each position of voxel (geometric information) and associated attributes are specified. Attributes typically describe voxel properties, including colors and reflectance. A PLY file generally describes a point cloud as a collection of vertices (and possibly edges and other elements), each accompanied by a set of properties defined in the header of file. Typically, a PLY file representing a point cloud contains a list of (X, Y, Z) triples for vertices, along with associated properties, as we can see below:

```
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
```

The above illustrated PC from the third frame of the "Ricardo9"sample file within the JPEG Pleno Database [14]. The file begins with a header that adheres to the conventions of the PLY format. The initial line identifies the file as a PLY format. The subsequent line declares the file format, which, aside from the discussed ASCII, could alternatively be in a binary format. Following this, the header outlines the elements from the object, with the first element being "vertex"and its count in the file, which amounts to 213,609. A series of property definitions follow, specifying the type and name of each property. Given the object in question is a point cloud, no further elements are listed beyond this point. The header concludes with an end_header line.

Subsequent to the header, the file lists the elements in accordance with the declarations from the header, mapping property values to each element type accordingly. In this

instance, the ensuing 213,609 lines detail the voxels from the point cloud. Each line describes a position from a voxel via (X, Y, Z) coordinates along with its color attributes (R, G, B) [15].

2.2 Signal compression

At the heart of *compression algorithms* lies a dichotomy: the *compression algorithm* (Encoder) and the *reconstruction algorithm* (Decoder). The former processes input data $\mathcal{X}$, producing a compressed counterpart $\mathcal{X}_c$ that embodies the original data in a reduced bit representation. Conversely, the Decoder reconstructs $\mathcal{Y}$ from $\mathcal{X}_c$, aiming to replicate the original dataset $\mathcal{X}$. This process bifurcates into *lossless* compression, where $\mathcal{Y} = \mathcal{X}$, and *lossy* compression, where $\mathcal{Y} \neq \mathcal{X}$ but typically achieves a higher compression rate.

The genesis of data compression algorithms entails two critical stages: *modelling* and *coding*. The former identifies and characterizes data redundancies within a model, while the latter encodes this model using a binary alphabet, often generating a *residual* that quantifies the deviation from the original data.

The key to evaluating compression methodologies are two metrics: the *compression rate* (CR) and *rate* (R), the former being the ratio of the original data size (S_o) to the compressed data size (S_c):

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

or, alternatively, expressed as a percentage reduction:

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

Generally, the *rate* is defined as the average number of bits required to represent a unit of information in the compressed format. In mathematical terms, for a given dataset, the rate can be expressed as:

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

where R is the rate, S_c is the size of the compressed data in bits, and L is the length of the dataset or the number of samples within it.

Particularly in the context of point clouds, which are comprised of three-dimensional data points representing the external surface of objects, the concept of rate adopts a specific form known as *bits per voxel occupied*. This measure reflects the average number of bits used to encode each voxel within the point cloud. Voxels, the three-dimensional

equivalent of pixels, serve as the fundamental unit of space in a voxelized point cloud. Therefore, the rate in this context is calculated as:

$$R = \frac{S_c}{N} \tag{2.4}$$

where N denotes the number of voxels occupied by the point cloud. This metric is particularly insightful for assessing the efficiency of point cloud compression algorithms, offering a direct measure of how compactly the three-dimensional spatial and attribute information is encoded.

In the realm of *lossy* compression, *distortion* measures the divergence between the original and reconstructed data, serving as an indicator of compression efficiency. Delving deeper into the principles underpinning data compression, we find in information theory the concept of *self-information*. Defined for an event A, with $P(A)$ representing the probability of A occurring, self-information ($i(A)$) quantifies the amount of surprise or new knowledge contributed by the occurrence of A. It is mathematically defined as:

$$i(A) = -\log_b P(A) \tag{2.5}$$

The choice of base b for the logarithm determines the unit of measurement: bits for $b = 2$, nats for $b = e$, and hartleys for $b = 10$. This logarithmic relationship underscores that events with higher probabilities contribute less new information (are less surprising), whereas those with lower probabilities contribute more (are more surprising).

For a series of independent events A_i within an experiment $\mathcal{S}$:

$$\bigcup A_i = S \tag{2.6}$$

the *entropy* (H) of the experiment, or the average self-information, is given by:

$$H = -\sum P(A_i) \log_b P(A_i) \tag{2.7}$$

Shannon's entropy encapsulates the average minimum number of binary symbols required to encode the output of a source, thus representing the theoretical limit of lossless compression efficiency.

Effective data modeling, whether through physical, probabilistic, or Markov models, is essential for approximating source entropy and devising efficient compression algorithms. Symbols from a given source data set form an *alphabet* $\mathcal{A}$, with binary sequences, or *codewords*, assigned to each symbol during the coding phase, constituting a *code*.

2.2.1 Geometry compression

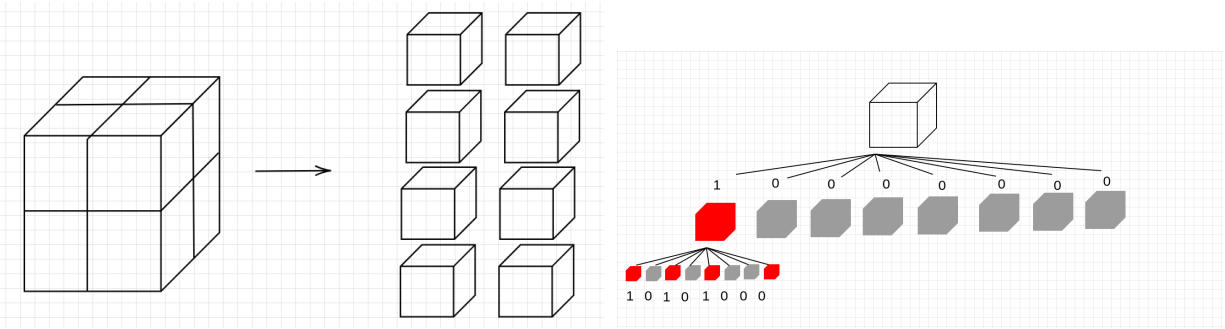
Building on the foundational concepts of signal compression presented earlier, it is appropriate to apply this understanding to the specific problem at hand. The discussion has elucidated that through a compression algorithm, data can be compacted into a file with fewer bits and later reconstructed. This algorithmic process bifurcates into two pivotal phases: modeling and coding. The chosen model informs the coding phase, where data source letters (from a defined alphabet) are translated into associated binary sequences, known as codewords (from a designated code). Once the algorithm is both modeled and implemented, assessing its performance becomes paramount. Key metrics for this analysis include entropy and, in the context of lossy compression algorithms, distortion. However, when focusing on point clouds, a vital question arises: what specific data is compressed, given its potential to encapsulate a wide array of information?

The utility of a point cloud, and the attributes deemed essential for encoding, may vary significantly depending on the application. Attributes such as color, geometry, and reflectance are often critical. For the algorithm discussed herein, the emphasis will be solely on geometry compression.

Geometry compression concerns itself with encoding the three-dimensional shape and contour of a point cloud. This involves identifying and exploiting similarities across different sections of the volume, the space occupied, and the voids within. By analyzing these characteristics, it is possible to uncover side information that, though not explicitly represented, can be inferred from the context of the volume. Moreover, certain information may be deemed redundant and safely omitted without detracting from integrity of the model — a principle applied not only in point cloud compression but also in image and video encoding.

Thus, geometry compression aims to efficiently represent the 3D shape of the point cloud, leveraging the information and redundancies inherent in the 3D object relative to the algorithmic approach adopted. It is crucial to recognize that geometry compression is not exclusive to point clouds but extends to other 3D representations, such as meshes, where additional volume-related information, like vertex connectivity, can be utilized.

A prominent example of a geometry compression algorithm is the octree approach [16], which recursively divides the point cloud into eight cubes until reaching the voxel level. This decomposition strategy, an extension of the 3D quad-tree [3], marks occupied volumes as '1' in the tree and empty volumes as '0'. These unoccupied spaces are treated as leaves, as illustrated in figures 2.1a and 2.1b [3], effectively capturing the spatial hierarchy and density of the point cloud.



(a) Unit cube divided into 8 sub-cube. (b) The volume encoding process.

Figura 2.1: The octree geometry compression approach illustration

2.3 CABAC

After a geometry compression has been applied to a volumetric object model, a substantial reduction in the number of bits required to represent its data is achieved. However, geometric compression alone often falls short of maximizing efficiency in data compression. To further optimize the representation of geometry from the point cloud, leveraging classic signal compression algorithms on the compressed geometric data becomes a crucial step. These classical algorithms facilitate the generation of two principal types of codes: *fixed-length* codes and *variable-length* codes.

Fixed-length codes are characterized by their uniformity; each symbol from the alphabet is represented by codewords that consist of an identical number of bits. This approach ensures simplicity and ease of decoding but may not always be the most space-efficient. Conversely, variable-length codes assign a unique bit length to each symbol based on its frequency or importance, allowing more common symbols to be encoded with fewer bits. This adaptability can significantly reduce the overall size of the compressed data, making variable-length codes particularly effective for further compressing the geometry of point clouds.

A pivotal example of a variable-length compression algorithm, employed in the proposed point cloud compression framework, is the *Context Adaptive Binary Arithmetic Coding* (CABAC). This sophisticated algorithm represents the forefront of variable-length coding, optimizing compression efficiency by dynamically adjusting the coding process based on the context of the data being encoded.

This section delve into the fundamentals of the Context Adaptive Binary Arithmetic Coder, outlining its basic implementation before transitioning to its application within the targeted algorithm for point cloud compression. Starting with an exploration of arithmetic coding principles, we will progressively build up to the context-adaptive aspects that distinguish CABAC. By adjusting codeword lengths in response to the statistical

properties of the data, CABAC achieves a high compression ratio, making it an invaluable tool in the realm of point cloud compression. The progression from elementary arithmetic coding to the sophisticated application of Context-Adaptive Binary Arithmetic Coding (CABAC) for point clouds elucidates the adaptation of traditional signal compression methodologies to address the unique demands of volumetric data. This adaptation is crucial for achieving efficient storage and transmission, while preserving the fidelity of geometric information.

2.3.1 Arithmetic Coding

Arithmetic coding stands as a variable-length coding scheme, particularly effective for encoding small alphabets, including those with highly skewed probabilities or binary sources, such as the geometry compressed bitstream discussed in this work. Its utility shines when the objective is to segregate the modeling and coding aspects within a lossless compression framework.

The essence of arithmetic coding lies in its unique approach to generating codewords for groups or sequences of symbols rather than for individual symbols in isolation. This methodology yields enhanced compression efficiency for the final encoded sequence. Arithmetic coding circumvents the need to create explicit codes for every possible sequence, which would otherwise result in an impractical proliferation of side data. Instead, it employs a singular identifier known as a *tag*. This tag serves as a unique binary code generated from a sequence of symbols, facilitating both encoding and decoding processes by allowing the original sequence to be accurately deciphered and reconstructed from the tag.

The **coding process** necessitates a distinctive identifier or tag for each sequence of symbols. An ideal candidate for such a tag is a decimal value within the interval $[0, 1)$, which, due to its infinite granularity, can uniquely represent each sequence. To map a sequence of symbols to a value within this interval, the cumulative distribution function (*cdf*) is utilized, effectively linking the random variable representation of the symbol to its probability.

Given an alphabet $\mathcal{A} = \{a_1, a_2, ..., a_m\}$ of a discrete source, and X as a random variable:

$$X(a_i) = i, \quad a_i \in \mathcal{A} \tag{2.8}$$

we define a probability model $\mathcal{P}$, representing the probability density function for the random variable:

$$P(X = i) = P(a_i) \tag{2.9}$$

and the cumulative density function as:

$$F_X(i) = \sum_{k=1}^{i} P(X = k) \tag{2.10}$$

The process of **generating the tag**, synonymous with encoding in arithmetic coding, iteratively narrows the interval within which the tag is generated as more sequence elements are processed. Initially, the unit interval $[0, 1)$ is divided into sub-intervals $[F_X(i-1), F_X(i)), i = 1, ..., m$. Each symbol a_i is then associated with a specific sub-interval. As symbols are encoded, the corresponding interval for each symbol, $[F_X(k-1), F_X(k))$, is further subdivided in proportion to the original interval, and the interval from the tag is correspondingly adjusted. This refinement continues with each new symbol received.

For instance, with a three-letter alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ where $P(a_1) = 0.5$, $P(a_2) = 0.2$, and $P(a_3) = 0.3$, the cumulative function values would be $F_X(1) = 0.5$, $F_X(2) = 0.7$, and $F_X(3) = 1$. Considering a sequence a_1, a_2, a_3, the intervals derived through the aforementioned algorithmic steps would be illustrated in Figure 2.2, demonstrating that the intervals generated are distinct and non-overlapping for different sequences.

For encoding, we denote a source sequence of length n as $(x_1x_2...x_n)$. The portion of the sequence encoded up to the *kth* element is represented as $\mathbf{x} = (x_1x_2...x_k)$. The lower and upper bounds of the interval where the tag resides at the *kth* element, denoted by $l^{(k)}$ and $u^{(k)}$ respectively, are defined as:

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

$$u^{(k)} = l^{(k-1)} + (u^{(k-1)} - l^{(k-1)})F_X(x_k) \tag{2.12}$$

The tag value, represented as $\overline{T}_X(\mathbf{x})$, is the midpoint of the final interval determined by the last symbol:

$$\overline{T}_X(\mathbf{x}) = \frac{u^{(n)} + l^{(n)}}{2} \tag{2.13}$$

The encoding process iteratively narrows the interval as symbols are processed, assigning a unique decimal tag for each sequence. This tag is then truncated and converted into a fixed-point binary representation, forming the encoded data.

Decoding mirrors the encoding steps, utilizing the tag to sequentially recover the original data sequence. The process involves recalculating intervals to ensure the tag

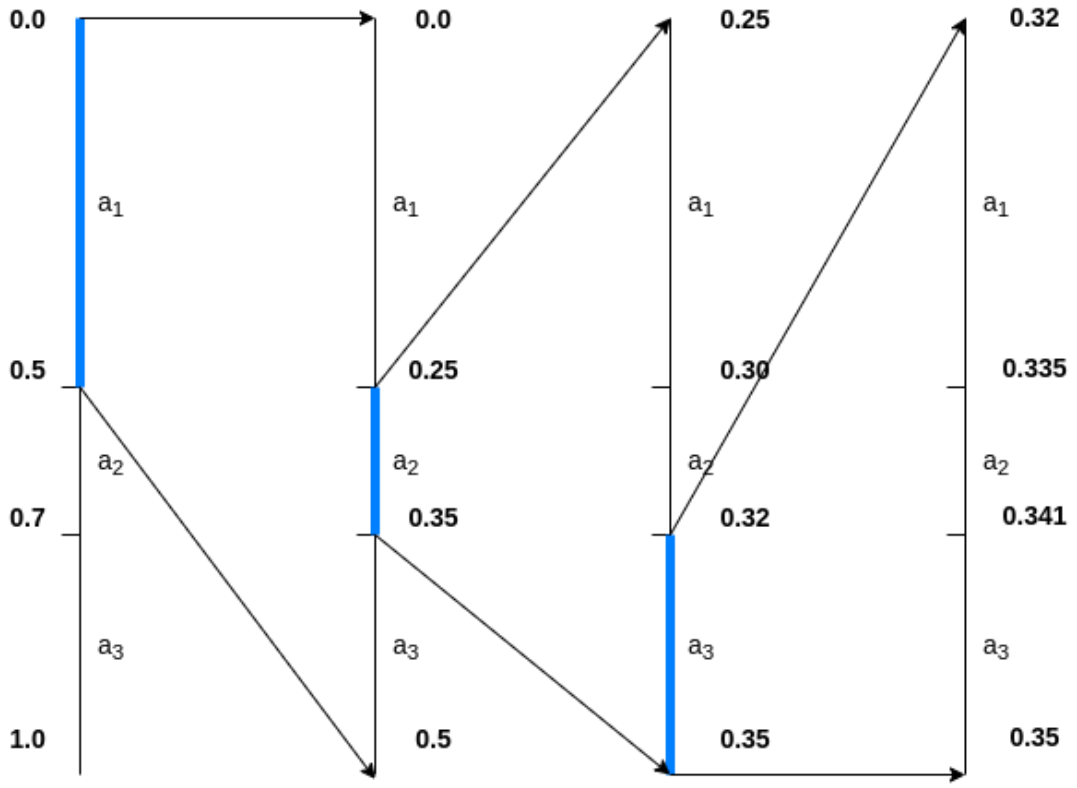


Figura 2.2: Arithmetic coding tag's interval generating for the sequence a_1, a_2, a_3.

value falls within the correct sub range for each symbol decoded. This approach ensures the reconstruction of the source sequence from the tag value.

Direct implementation faces challenges due to the finite precision of computer representations, leading to potential interval convergence due to truncation. To address this, intervals are rescaled as needed, maintaining synchronized and incremental encoding. Three scenarios trigger rescaling:

1. The interval lies entirely within the lower half of the unit interval $[0, 0.5)$.
2. The interval lies entirely within the upper half of the unit interval $[0.5, 1.0)$.
3. The interval straddles the midpoint of the unit interval.

Rescaling remaps the sub-interval to a new interval ranging $[0, 1)$, sending corresponding bits to the decoder and adjusting the interval accordingly:

1. For the lower half, send bit 1 and double the interval: $E_1(x) = 2x$.
2. For the upper half, send bit 0 and shift and double the interval: $E_2(x) = 2(x - 0.5)$.
3. Continue processing without rescaling for intervals straddling the midpoint.

This rescaling ensures that arithmetic coding remains effective even within the limitations of computer arithmetic, optimizing the balance between compression efficiency and computational feasibility.

2.3.2 Context Arithmetic Coding

A significant enhancement can be incorporated into arithmetic coding through the concept of *contextual information*. To elucidate, consider the following sequence:

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

Analyzing the frequency of a_2 within this sequence reveals its occurrence constitutes only 25% of the total. However, by considering the preceding symbol, we can ascertain a more precise probability of a_2's occurrence. Notably, a_2 follows either a_1 or a_2. The probability of a_2 following a_1 is 33%, while it escalates to 66% when preceding another a_2.

This increased accuracy in predicting the likelihood of a_2, given the knowledge of the preceding symbol, introduces a layer of *context*. Incorporating contextual information allows for a more nuanced probability estimate, enhancing data compression efficiency. By leveraging the relationships and frequencies of symbols already encoded/decoded, the algorithm gains an augmented capacity for predicting subsequent symbols based on preceding ones. This contextual approach not only enriches the predictive power of the model, but also underscores the adaptability and sophistication achievable in data compression strategies.

2.3.3 Context Adaptive Arithmetic Coding

In the example provided, we have assumed that the context probability table is readily available and computed. However, in most practical scenarios, this information is not initially accessible to the encoder. Consequently, the algorithm must adapt to newly learned distributions as the encoding progresses. A straightforward strategy involves initializing the table with all elements as counters set to 1. Initially, this implies limited knowledge about the source. As each symbol is encoded, the counter associated with its context is incremented. This mechanism is mirrored in the decoder, ensuring that after each symbol is encoded, the counter is updated accordingly.

Consider a source with an alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ tasked with decoding the message:

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

At the outset, the context table is initialized as follows, where each row represents a specific context:

Tabela 2.1: Initial Context Table

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

Following the encoding of the first symbol a_1, the table remains unchanged, as it is the initial symbol without preceding context. After encoding the second symbol a_1, the table is updated as shown in Table 2.2. This process is repeated until all symbols are encoded, resulting in the final context table presented in Table 2.3. This iterative process ensures that all contexts are updated, with probabilities adapting to each symbol encoded, thereby refining predictive power of the algorithm, based on previously encountered symbols.

Tabela 2.2: Context 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

Tabela 2.3: Context 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

2.3.4 Context Adaptive Binary Arithmetic Coding

Having discussed the basics of arithmetic encoding, we are well-prepared to introduce the **Context Adaptive Binary Arithmetic Coder (CABAC)**, pivotal for encoding the binary point cloud geometry in the S3D approach. CABAC proves to be advantageous not only for point clouds but in many scenarios where the alphabet is binary, such as bilevel documents or the binary representation of nonbinary data seen in H.264 CABAC. The primary benefit of a binary alphabet lies in its simplified probability model, requiring only

a single value to represent the probability of one symbol. The probability of the alternate symbol is then the complement of this value. This simplification facilitates the use of multiple contexts in the encoding process, significantly enhancing compression efficiency.

CABAC Mechanism

The CABAC algorithm combines the *Context Adaptive* approach with binary arithmetic coding. Initially, a word length m is chosen to map the interval $[0, 1)$ to 2^m binary words, establishing a direct correspondence between decimal and binary representations.

Given the unpredictability of message length, and adhering to an adaptive case strategy, the choice of m is made independent of message length. Sayood demonstrates that a word of length m accommodates a total count of 2^{m-2} or less, necessitating interval rescaling as the symbol count nears 2^{m-2}. This rescaling involves halving all numbers and rounding up to ensure no value is diminished to zero, a measure that refreshes the count table to better reflect the local statistics of the source.

Interval Computation and Updates

With a binary alphabet, interval computation simplifies to updating one endpoint and the interval size, denoted by:

$$A^{(n)} = u^{(n)} - l^{(n)} \tag{2.14}$$

The tag of the sequence is then the binary representation of $l^{(n)}$, with symbols treated as More Probable Symbol (MPS) and Less Probable Symbol (LPS) rather than direct binary values. The probability of an LPS occurrence in context C is q_c, influencing the interval update equations for MPS and LPS occurrences respectively. The mapping of interval values to binary words, and the significance of the most significant bit (MSB) in determining the half of the interval, introduce specific operations E_1, E_2, and E_3 to manage encoding efficiently:

1. If the MSB from both bounds is equal, a left shift is performed. This operation corresponds to E_1 or E_2 based on the value of the MSB: 0 for E_1 and 1 for E_2, effectively doubling the size of the interval or shifting it to accommodate the new value.

2. If the 2nd MSB from $u^{(n)}$ is 0 and from $l^{(n)}$ is 1, operation E_3 is engaged, shifting each bound 1 bit to the left and then complementing the MSB. This ensures the interval is adjusted without losing precision.

Consider n_j as the amount of the symbol j occurrence in a sequence of length $Total\_count$. Besides, considering Cum_count as:

$$Cum\_count(k) = \sum_{i=1}^{k} n_i \tag{2.15}$$

The updated bounds are calculated as follows:

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_count(x_n - 1)}{Total\_count} \right\rfloor \tag{2.16}$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_count(x_n)}{Total\_count} \right\rfloor - 1 \tag{2.17}$$

Decoding mirrors these steps, with t^* determined by:

$$t^* = \frac{(t - l + 1) \times Total\_Count - 1}{u - l + 1} \tag{2.18}$$

where t^* is the tag value. This process ensures the efficient reconstruction of the original message.

2.4 Silhouette 3D

Having established a foundational understanding of signal compression and the *CABAC*, an instrumental algorithm for this work, we now shift our focus to the primary algorithm responsible for point cloud compression. This algorithm draws inspiration from existing work but with notable distinctions. Before delving into the geometry compression part of our algorithm, it is essential to explore other point cloud geometry encoders for context.

One of the primary codecs for point cloud compression is based on an octree structure, which involves recursively dividing the point cloud into eight smaller cubes that comprise the entire volume [16]. The MPEG G-PCC geometry compression builds on the octree concept, applying an arithmetic coder to enhance the geometry compression. Other codec variations extend the octree approach with various arithmetic encoders [3]. An alternative method in the codec of M. Krivokuca et. al [17] utilizes volumetric functions based on *B-spline wavelets* to encode the geometry and attributes of point clouds. Additionally, strategies employing deep learning techniques represent another innovative direction for point cloud compression [5]. These methods leverage the ability of deep learning models to abstract complex data structures, including point clouds, into a more manageable form, often treated as sequences of symbols for compression.

The S3D algorithm introduces a novel approach by focusing on the silhouettes of point cloud slices for geometry coding. Unlike previous methods that emphasize volumetric properties and patterns directly from raw data, S3D leverages binary images of silhouettes as the basis for compression. This distinctive feature makes binary image compression techniques particularly suitable for S3D, resulting in efficient compression rates [8] and outperforming G-PCC v7.0 performance. The following section explores the theoretical underpinnings and methodologies employed in the S3D algorithm.

2.4.1 Silhouette

The S3D algorithm employs two primary approaches to determine the most effective compression method: the *dyadic decomposition* and the *single mode*. Both approaches adhere to the core concepts and elements proposed by the algorithm.

Central to these approaches is the utilization of the point cloud geometry as a 3D occupancy array $N \times N \times N$, where occupied coordinates are marked as 1, and unoccupied as 0, effectively treating it as a 3D Boolean array $G(x, y, z)$. By slicing the point cloud along a specific axis and projecting the occupied points within each slice onto an $N \times N$ image, we generate what can be likened to a *silhouette* of the region from the point cloud. A silhouette is thus defined as:

$$I(i,j) = silhouette(G, axis, iStart, iEnd) = \begin{cases} \sum_{n=iStart}^{iEnd} G(axis, start, end) & \text{if } axis = x \\ \sum_{n=iStart}^{iEnd} G(start, axis, end) & \text{if } axis = y \\ \sum_{n=iStart}^{iEnd} G(start, end, axis) & \text{if } axis = z \end{cases} \tag{2.19}$$

Where the summation is performed via a logical **OR** operation, merging all slices in the interval $[iStart, iEnd]$ into a single bitmap image $I(i, j)$. This process is visualized in Figures 2.3 and 2.4, illustrating the sliced point cloud and its respective projected silhouettes.

2.4.2 Dyadic Decomposition

Having established silhouettes as elements that describe the point cloud geometry from its slices, the algorithm employs a recursive basis. First, a deep analysis into the *dyadic decomposition* will be conducted, a primary approach utilized in the S3D algorithm. Initially, the point cloud is segmented into two smaller intervals, each further divided in two, adhering to the *dyadic decomposition* principle where interval ranges are halved from

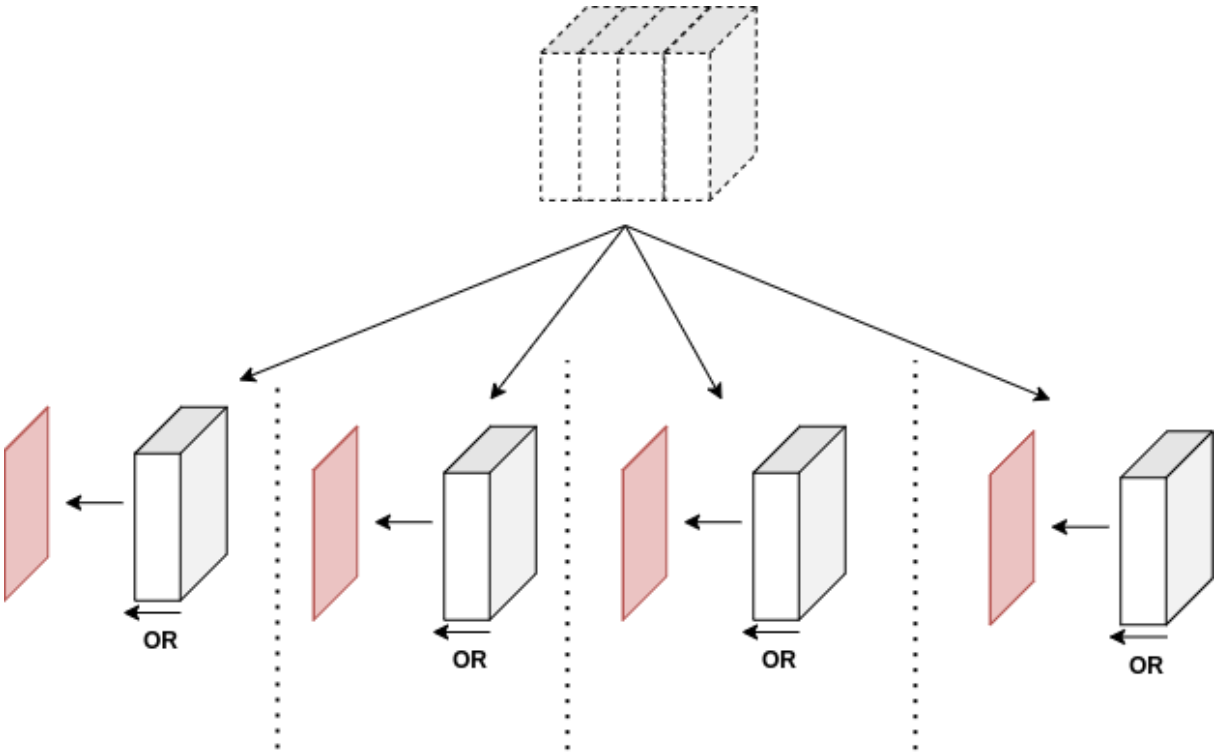


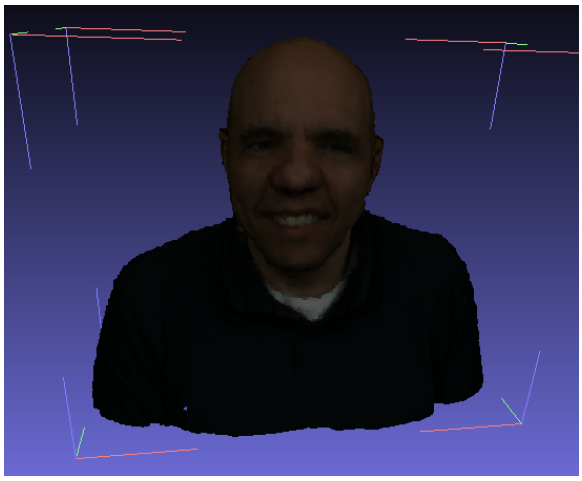
Figura 2.3: Point cloud sliced and its respective silhouettes projected

the original slice. This recursive halving continues until a slice contains no points or reaches an atomic width of 1, becoming essentially a bitmap image.

This recursive process manifests as a binary tree, where each node represents a slice of the point cloud. From these slices, $N \times N$ silhouettes are projected, as depicted in Fig. 2.5 [9], with each node signifying a slice and the red shading indicating a silhouette projection along the vertical axis.

Given this binary tree of silhouettes that describe the point cloud at each interval, the geometry compression, rooted in *silhouette decomposition*, begins by encoding the bitmap image of each tree node to transmit the images that comprise the entire point cloud. This approach leverages a specific characteristic of the silhouette generation process regarding unoccupied voxels: if a silhouette pixel is blank, it implies the absence of occupied pixels across all contributing slices for that coordinate. This inference is drawn from the **OR** operation employed during silhouette creation, ensuring the occupation of the pixel in the projection if it is occupied in at least one slice.

In the tree context, no subtree generated from a node silhouette will contain occupied pixels where the parent node does not. Furthermore, this decomposition process capitalizes on the similarities between adjacent node silhouettes—the parent node and its sibling, or "brother node,"leading to the nomenclature *dyadic decomposition*. It is noteworthy that this process excludes the root node due to its lack of a parent or sibling node.



(a) ricardo9 point cloud rendered

(b) ricardo9 silhouette along axis X

(c) ricardo9 silhouette along axis Y

(d) ricardo9 silhouette along axis Z

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

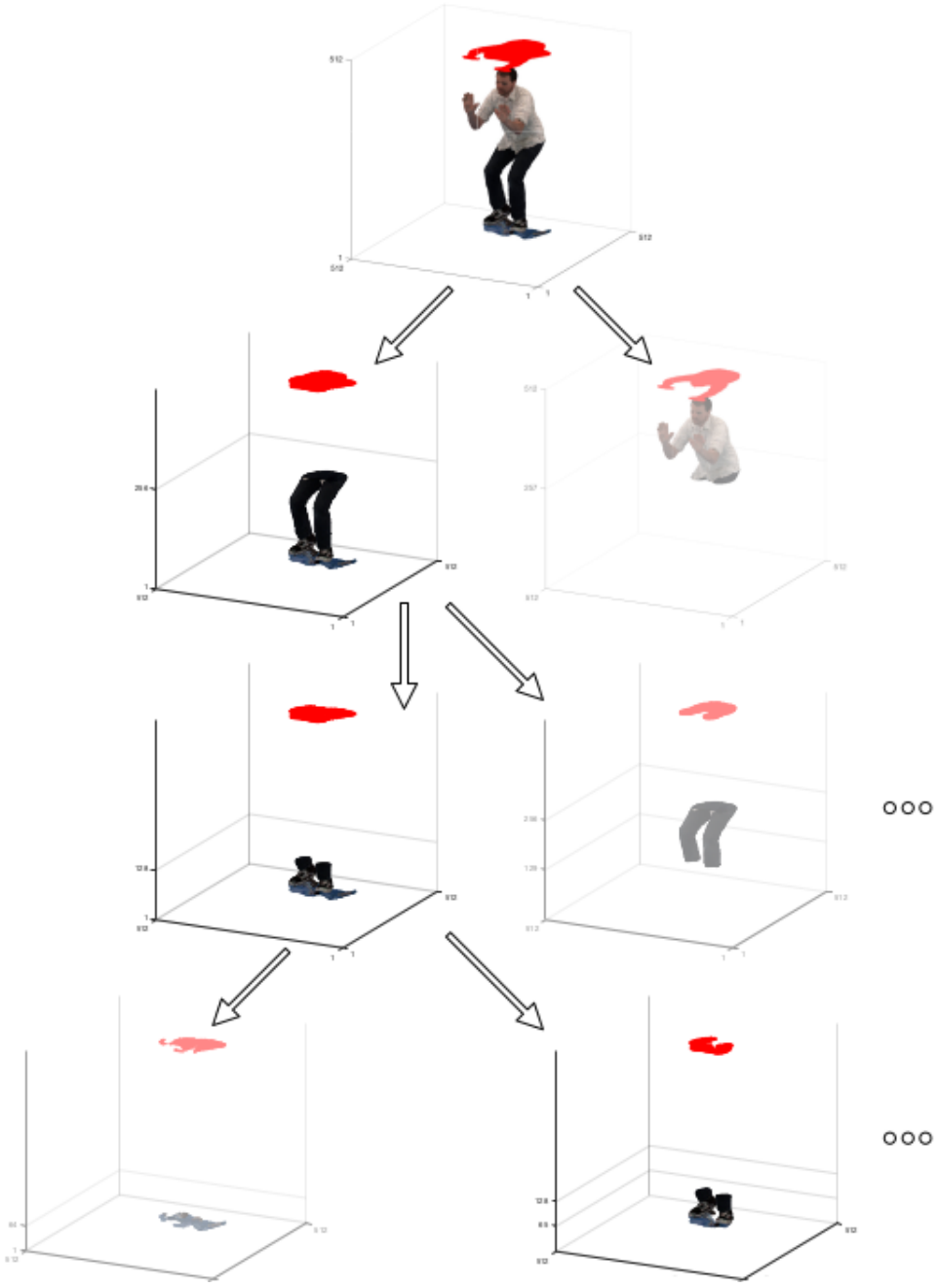


Figura 2.5: Binary tree derived from the point cloud recursive slices.

For a node in the tree Y_C, along with its child nodes Y_L (left) and Y_R (right), the transmission of both children assumes Y_C has already been transmitted, following these steps for transmission:

1. For transmission of Y_L, utilizing the parent image Y_C as a mask, only the bits where Y_C is 1 are sent from Y_L.
2. For transmission of Y_R, considering both the parent image Y_C and Y_L as masks, only the bits where both Y_C and Y_L are 1 are sent from Y_R.

This decomposition approach is viable because $Y_C = Y_L + Y_R$ under an **OR** operation. The scenario where Y_C is 0 implies that both Y_L and Y_R are also 0. Given prior transmission of Y_C, these values can be inferred for both child silhouettes, hence transmitting only where Y_C is 1. The rationale for the second decomposition step, given transmission from Y_C and Y_L, is that the bits where Y_C is 1 and Y_L is 0 necessitate Y_R to be 1, thus only transmitting bits where both parent and right child are 1.

Consider the illustration in Fig. 2.6 for a clearer understanding.

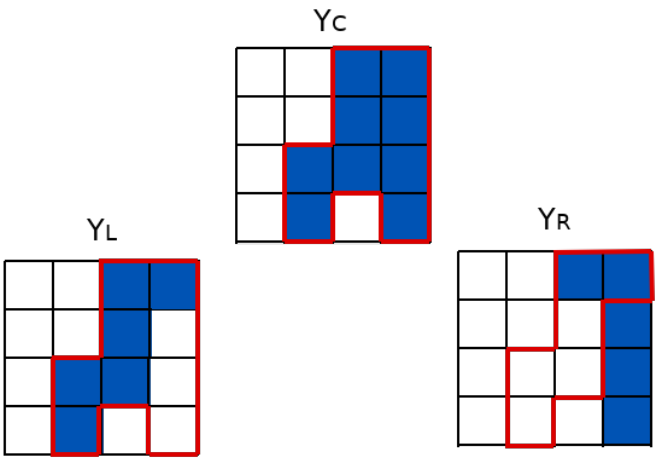


Figura 2.6: A simple silhouette decomposition illustration

Blue pixels denote occupied areas, white pixels represent empty spaces, and the red contour outlines the transmission-required bits, delimited by the mask of occupied pixels in either the parent or left node silhouette. Fig. 2.7 expands this decomposition across three levels, with silhouette 1 as the root node. Only the root silhouette is transmitted in full, with subsequent silhouettes transmitted via decomposition. The bitstream sent from each node is:

1. 0011001101110101

2. 111011010

3. 001111

4. 1010

5. 110000

6. 11100

7. 111

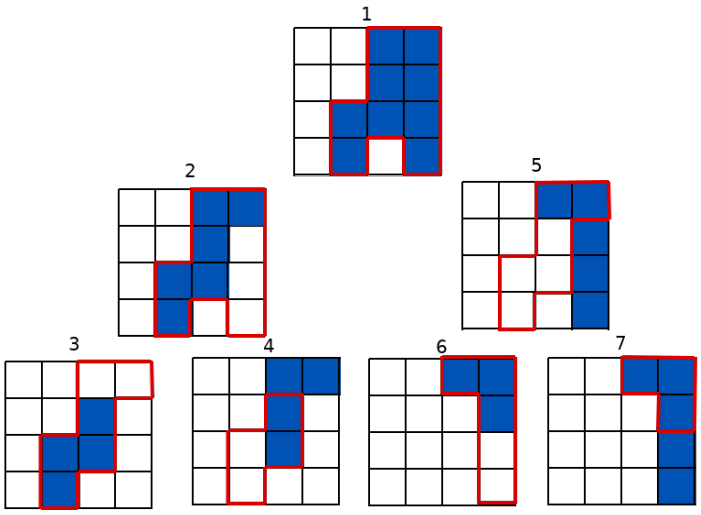


Figura 2.7: A silhouette decomposition example with 3 levels.

The final bitstream, following the number ordering, becomes:

00110011011101011110110100011111010110000011100111

This sequence occupies 49 bits, compared to sending all four 4×4 images, which would occupy 64 bits. It is important to note that the silhouette decomposition process follows a *preorder tree traversal*, encoding from the parent node to the left child node and then to the right child node, recursively. Thus, the images from the point cloud are transmitted in this order.

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

The CABAC has 16 bits of precision, and all its contexts are initialized only once with a value of 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 pixel contexts are called **2D contexts**. The image Y_L, following the tree notation, is encoded with the 2D contexts, 5 pixels, plus 9 pixel contexts from an additional image: the Y_C. These are referred to as **3D contexts**, as shown in Fig. 2.8 (b) [8]. Finally, the image Y_R is encoded using the 2D contexts, with the additional 3D contexts provided by the Y_L pixels. This encoding process is illustrated in Fig. 2.9. This procedure is repeated until all nodes are covered, following a preorder traversal, culminating in the final encoded bitstream.

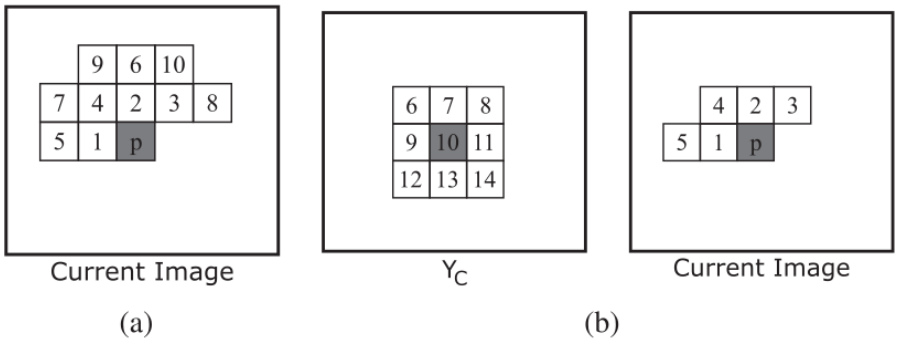


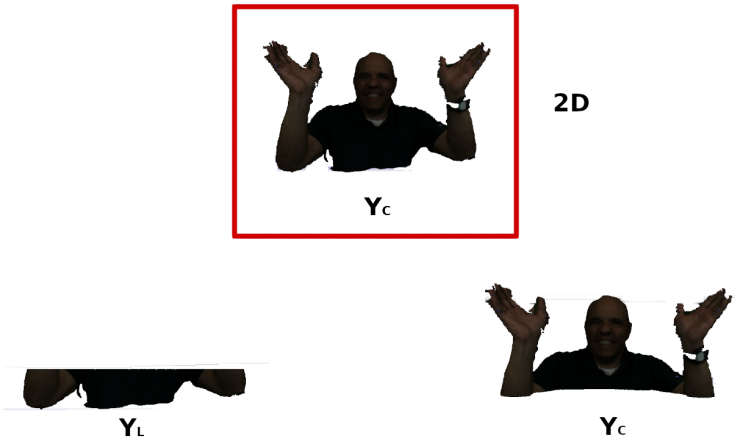
Figura 2.8: Contexts used to encode pixel p: (a) 2D Contexts and (b) 3D Contexts

The decoding process mirrors the encoding steps. Initially, the bitstream is decoded using CABAC. Armed with an understanding of the tree traversal sequence, the decoder can identify which image from the tree is being decoded and, consequently, apply the appropriate 3D contexts derived from the pixels of external images. Once the geometry bitstream is recovered, it becomes feasible to reconstruct the silhouettes from the point cloud that populate its occupancy voxels. Given that the leaf nodes represent slices with a width of 1, their alignment and combination effectively reconstitute the entire point cloud, as illustrated in Fig. 2.10. Through this process, we can successfully restore the original point cloud without any loss of voxels.

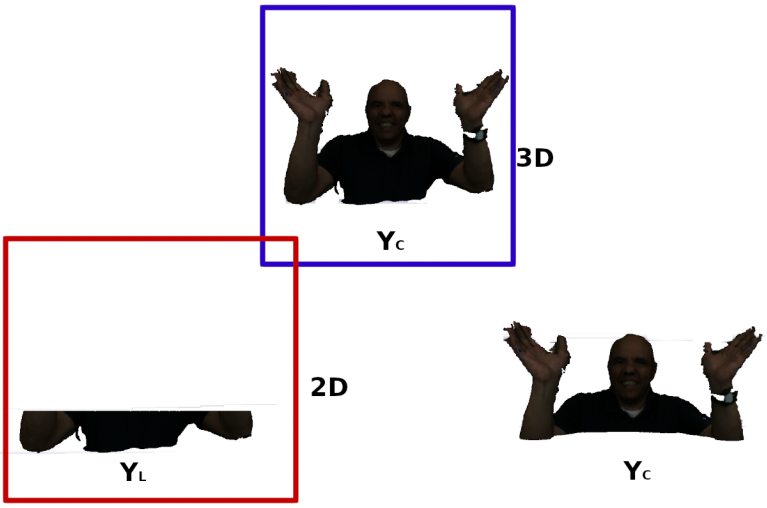
2.4.3 Single Mode

With an understanding of how Dyadic Decomposition (DD) operates, grasping the mechanics of the Single Mode (SM) mechanism becomes simpler, as it builds upon the same foundational encoding principles.

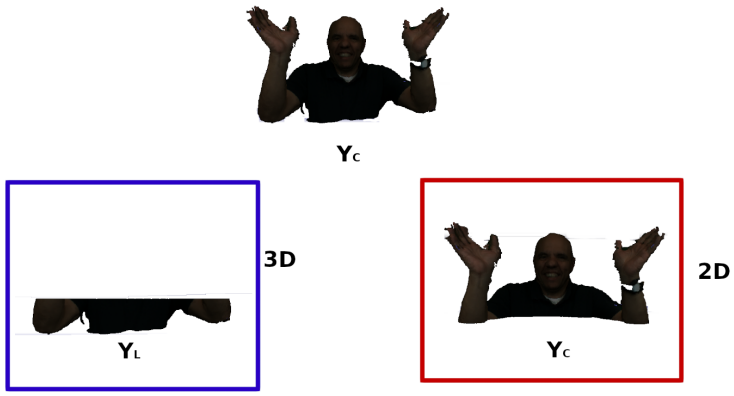
Consider a point cloud segmented along an arbitrary axis a from the set $S = \{x, y, z\}$, resulting in a slice of length L. This process generates L slices, or 'children', from the point cloud, as depicted in Fig. 2.11. Each point cloud slice, and consequently its silhouette I_f



(a) Y_C encoding



(b) Y_L encoding



(c) Y_R encoding

Figura 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.

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

(termed the 'father silhouette'), alongside the silhouettes I_c^i generated from each slice (the 'children silhouettes'), where $0 \leq i < L$, form the basis for the Single Mode encoding.

The encoding begins with the first child silhouette, utilizing contexts exclusively from I_f. Subsequent child silhouettes I_c^i are encoded by employing contexts from both the father silhouette and the previously encoded child silhouette I_c^{i-1}. This process continues through to the final silhouette, which is encoded using contexts from I_c^{i-1} and a mask generated by iteratively performing an OR operation across all previously encoded children, denoted as $\sum_{n=0}^{L-2} I_c^i$. Consequently, the Single Mode bitstream is composed of the CABAC-encoded silhouettes of the L point cloud slices.

2.4.4 Best choice

Exploring the two encoding strategies reveals distinct characteristics of each. The *Dyadic Decomposition (DD)* unfolds as a recursive process within a tree structure, methodically dividing the point cloud into halves at each level until achieving a unitary width. Conversely, the *Single Mode (SM)* adopts a straightforward approach, encoding all one-width slices of a point cloud simultaneously, culminating in the final bitstream. This can be envisaged as a tree with just two levels: the root representing the entire silhouette of

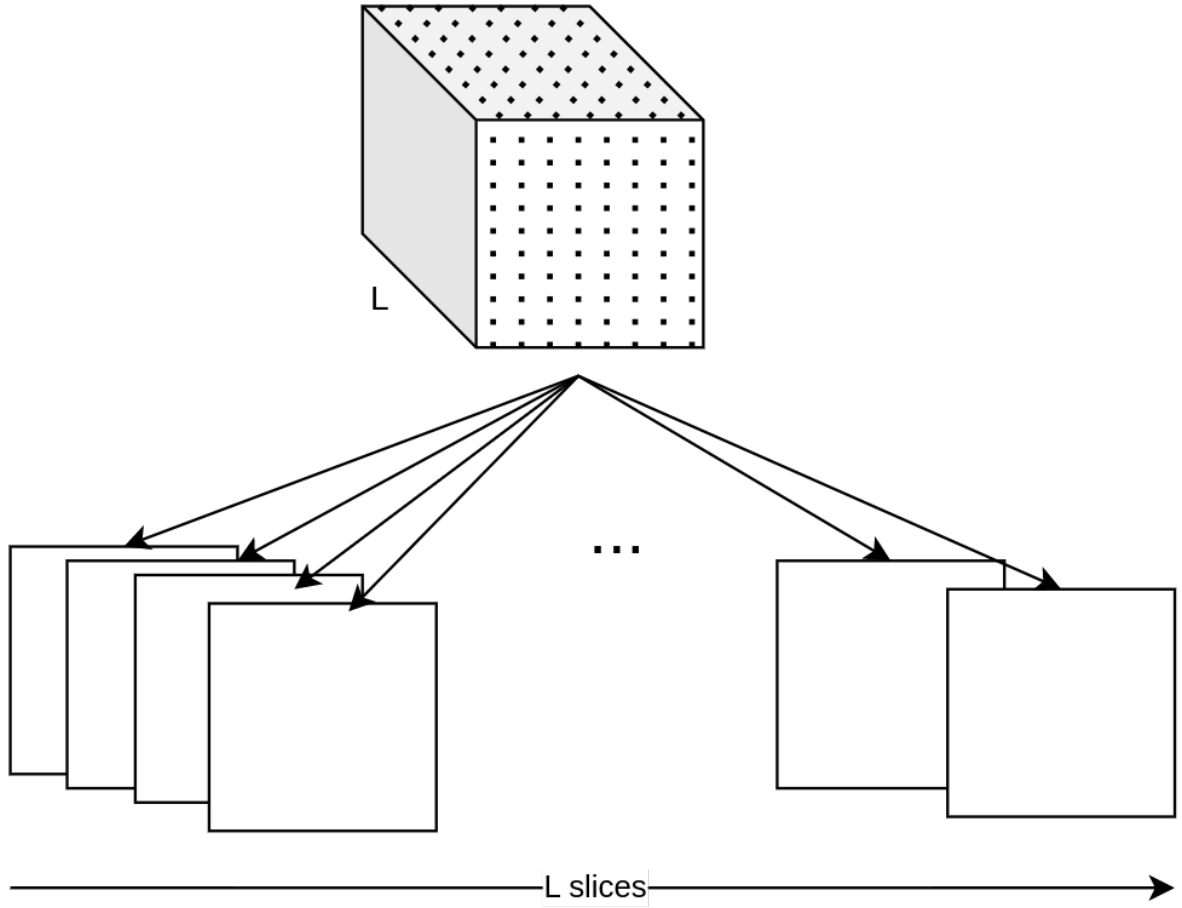


Figura 2.11: Representation of the single mode operations - this figure illustrates a point cloud of length L, resulting in L slices during the single mode operation.

point cloud and its direct children corresponding to the leaf nodes—each a slice of width 1.

From these perspectives, each algorithm can be succinctly summarized: *DD* is recursive, requiring successive applications on increasingly smaller segments, whereas *SM* operates in a singular, comprehensive step, immediately yielding its output upon application.

The S3D algorithm, thus, can be visualized as originating from a *DD* recursion tree, where at any node, an application of *SM* might prune the tree. This conceptualization provides a holistic view of the S3D's adaptability, employing the most effective encoding combination at each step for optimal performance.

Through this metaphorical explanation, the operational essence of S3D is unveiled. Initially, both *SM* and *DD* are evaluated to determine which approach offers superior compression performance. Should *SM* emerge as the preferred choice, the algorithm concludes promptly. Alternatively, selecting *DD* opens avenues for further decision-making in subsequent subtrees, indicating a potentially intensive computational process due to

the exhaustive evaluation of all feasible combinations.

Consequently, the S3D bitstream embodies the most efficient compression pathway among all considered options. To assist the decoder in navigating these choices, flags are embedded within each bitstream segment, signaling the encoding strategy employed for each subtree. This methodology promises significant compression rate enhancements by leveraging optimal choices. Nonetheless, the associated computational cost of discerning these options must also be acknowledged.

2.5 Threads

This work proposes an algorithm that utilizes *multi-threading* programming to leverage the capability of encoding the point cloud decomposition tree subtrees concurrently, aiming to enhance the algorithm's time performance. To grasp the mechanism of this algorithm, a review of some pivotal concepts in this domain is necessary.

2.5.1 Multi-threading

As described by Tanenbaum [18], a thread is essentially a basic unit of CPU utilization, comprising an ID, a program counter, a register set, and a stack. It operates as a "lightweight process."Being an independent execution flow, multiple threads can run in cooperation within a single process, sharing the resources allocated to that process. Hence, a thread represents an execution path within an operation of process. A simple process functioning with a single execution flow is recognized as a *single-threaded* program. Conversely, if a process executes more than one concurrent flow, sharing resources like memory, stack, and files, it is considered *multi-threaded*.

Introducing multiple execution flows might raise concerns about increased code complexity, especially when threads access a shared resource simultaneously, potentially leading to inconsistent states. The question arises: why pursue multi-threading given the complexity of managing concurrent access to shared resources? The primary motivation for integrating concurrency is to fully utilize the capabilities of the computing device. In scenarios where algorithm components are independent, a single-threaded implementation would force these components to execute sequentially due to the singular flow of execution. By adopting multi-threading, different parts of the algorithm can be processed in parallel, even as computations continue in other threads. This approach ensures continuous CPU engagement, contrasting with single-threaded models where CPU resources may remain underutilized.

Another key advantage of threading is the shared resources among threads, particularly relevant when processing large point clouds on devices with capable CPUs and substantial

memory. A standalone, single-threaded process might not efficiently manage memory or CPU resources. However, threads, being lightweight, allow for parallel computation without significantly increasing memory usage. Despite the added complexity of managing concurrency, the benefits of enhanced computational throughput and efficient resource utilization justify the multi-threaded approach.

Race Condition

When two execution flows operate concurrently, sharing common resources without synchronized access, unexpected outcomes can arise. A classic scenario involves reader and writer processes accessing shared memory. Imagine two processes reading from, and one process writing to, this shared memory. The sequence of their operations might unfold in several ways:

1. The writer updates the shared memory with new data, denoted as α, followed by both readers accessing this updated information, reading α
2. One reader accesses the memory first, reading the old data γ, before the writer updates it to α; subsequently, the second reader accesses the updated data, α.
3. Both readers access the memory before the writer has a chance to update it, reading the old data γ; the writer then updates the memory to α

In scenarios where readers must access the latest data, the latter two cases present inconsistencies, with at least one reader obtaining outdated information, γ. The second scenario further complicates matters by introducing a discrepancy in the data read by the two processes, casting doubt on the reliability of the data.

Critical Section

The root of such issues lies in multiple processes accessing shared memory without coordinated operation management. If a writing process initiates, no other operation should occur simultaneously on that memory. This principle also applies to reading processes; if a read occurs while another process writes, the reader may receive a mix of old and new data, leading to inconsistency among processes accessing the memory. To ensure proper behavior, processes must mutually exclude each other from accessing the shared resource during operation. The program code segment where the shared resource is accessed is termed the *critical section*. Guaranteeing that only one process or thread executes within the critical section at any time upholds *mutual exclusion*. A straightforward and effective solution employed in this work is the use of a **mutex**.

Mutex

A *mutex* (mutual exclusion lock) is a mechanism designed to prevent race conditions by ensuring one process/thread can access a shared resource at a time. Originating from the semaphore concept by E. W. Dijkstra, the mutex addresses concurrency issues such as the **producer-consumer problem** through *lock* and *unlock* operations and two states: *locked* and *unlocked*.

Locking a *mutex* (mutual exclusion lock) is a critical operation in concurrent programming. When a thread attempts to lock a mutex that is already in a locked state, it enters a wait state. This mechanism prevents multiple threads from accessing a shared resource simultaneously, which could lead to data corruption or inconsistent states. The thread that has locked the mutex effectively has exclusive access to the shared resource it protects.

During this waiting period, the thread is suspended, consuming minimal resources as it awaits the release (unlocking) of the mutex. This suspension ensures that other threads, including the one holding the mutex lock, can continue execution, potentially leading to the eventual unlocking of the mutex.

Once the mutex is unlocked by the thread that originally locked it, the waiting thread (or one of the waiting threads, if multiple) is awakened and given the opportunity to acquire the mutex lock. This transition from a locked to unlocked state, followed by the immediate locking by a previously waiting thread, maintains a controlled and orderly access to the shared resource. Notably using standard C++ library *<threads>*, if multiple threads are waiting, the release of one waiting thread upon unlocking does not follow a specified order according to its documentation.

The effectiveness of mutexes in enforcing mutual exclusion hinges on the **atomicity** of lock and unlock operations, ensuring these actions are indivisible and executed to completion before any other thread can perform the same operation. This atomic nature guarantees the consistent state of the mutex. Fig. 2.12 illustrates the behavior and state transitions of a mutex in managing access to shared resources.

This chapter reviewed key concepts from the signal compression field, with a focus on voxelized point clouds and their geometry. It also covered the theory behind arithmetic encoding and the Context-Adaptive Binary Arithmetic Coding (CABAC), which is central to the silhouette encoding in S3D. Next, the foundational principles of S3D were explained, as they are crucial for the algorithms proposed in the following chapter. Finally, the theory of concurrent programming was introduced to enhance understanding of the role of threads in algorithm implementation.

In the next chapter, two main groups of algorithm proposals will be presented: first, the S3D Block Mode and S3D Inverted Mode based on block decomposition, followed

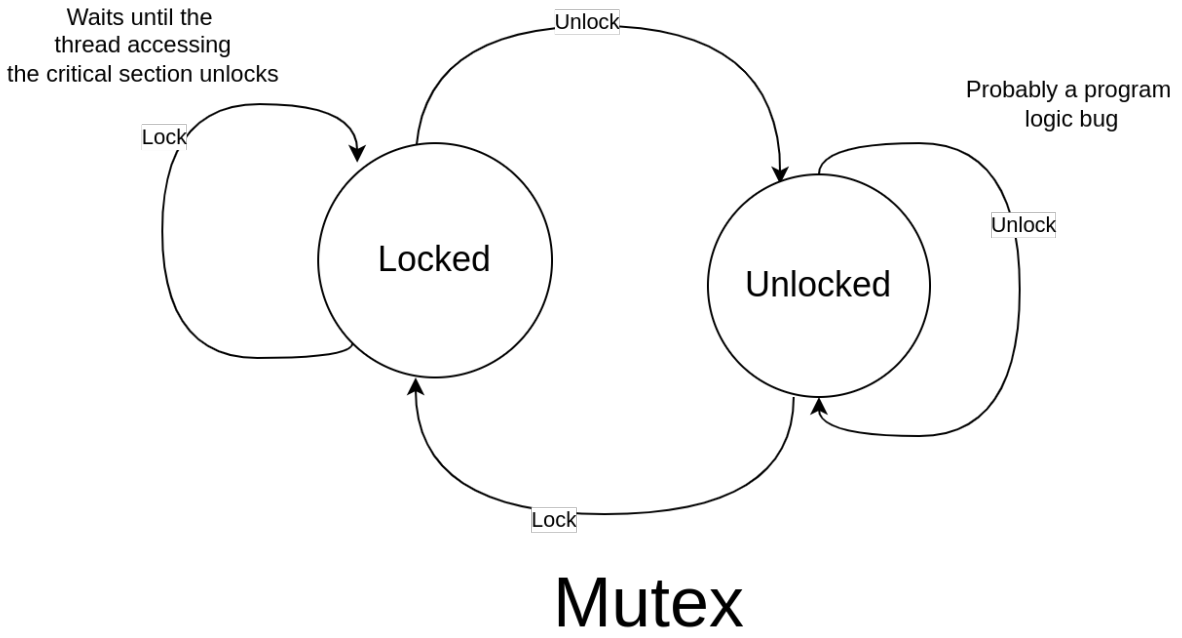


Figura 2.12: State diagram summarizing the mutex variable behaviour given the mutex variable operations.

by the multi-threaded S3D-S and S3D-ST. Implementation details will also be discussed, including a unified header model for standardization in future versions. Additionally, a new order of 3D pixel context selection will be proposed, setting the stage for further research into S3D codec variants implementation.

Capítulo 3

Proposed Algorithms

In the preceding section, we explored the essential subjects within the domain of point cloud compression, delving into the core principles and various representations of point clouds. Subsequently, we touched on key notions within the field of signal compression and its relevance to point cloud compression. We then provided an in-depth explanation of how an arithmetic codec operates, specifically the Context-Adaptive Binary Arithmetic Coding (CABAC) model that influenced the codec used in this research. Following this, we introduced the geometry codec central to our study - the Silhouette 3D. Moreover, we examined concurrent programming principles through the lens of multi-threading, gaining insights into its fundamental concepts and the potential challenges it poses for this research.

In this section, we will provide a detailed discussion of the algorithms proposed in this study, emphasizing their development process, underlying motivations, as well as their benefits and limitations. Furthermore, we will address the updates made to the S3D algorithm, focusing on its header design and silhouette-context-based arithmetic coding.

3.1 Multi-threaded S3D

One of the standout features of the S3D (Silhouette 3D) method in geometry representation and encoding, as it navigates through the point cloud, is the modular nature of the encoding components. Specifically, when we diagrammatically represent the S3D decomposition process using a tree structure, as shown in Figure 3.1, a repeating pattern emerges. This pattern involves context-based encoding among a set of three nodes: a parent and its two child nodes, with one on the left and the other on the right, as illustrated in Figure 3.2. This trio of nodes can be seen as a foundational element that, through repeated occurrence, forms the entire decomposition tree, as depicted in Figure 3.3. Analyzing the encoding process for each triad reveals that, theoretically, the enco-

ding of each is independent of the others, aside from the contextual information stored alongside the pixel encoding for each silhouette, which could enhance the efficiency of the arithmetic codec. Additionally, the silhouette decomposition process utilizes only the known occupied and unoccupied pixels from the parent node or its left sibling, making the process self-contained within each triad.

This perspective allows us to view the decomposition tree as composed of discrete, independent components, as exemplified in Figure 3.4. Each subtree at a given level can be seen as an autonomous segment of the whole. Given its recursive structural representation—a tree made up of smaller trees, echoing the principle of a fractal—this design was deemed an intriguing subject for examination. Specifically, this structure laid the groundwork for the development of multi-threaded versions of the S3D algorithm, as proposed in the S3D-subtree (S3D-S) and S3D-subtree top (S3D-ST) models [12].

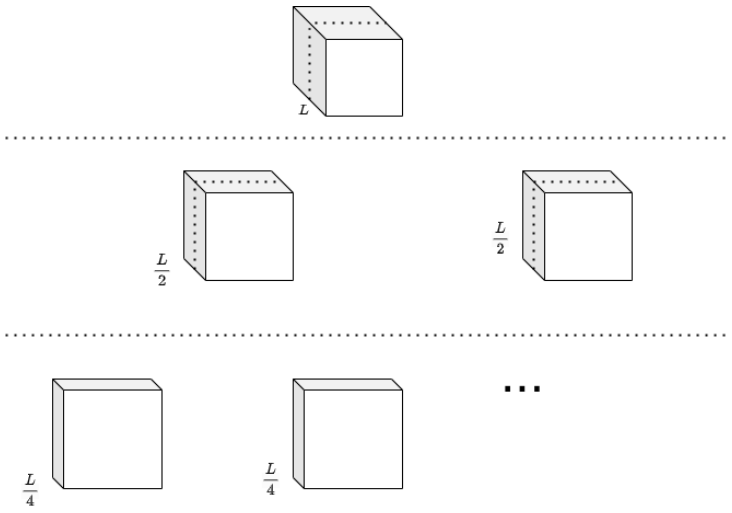


Figura 3.1: Representation of the decomposition tree from the dyadic decomposition.

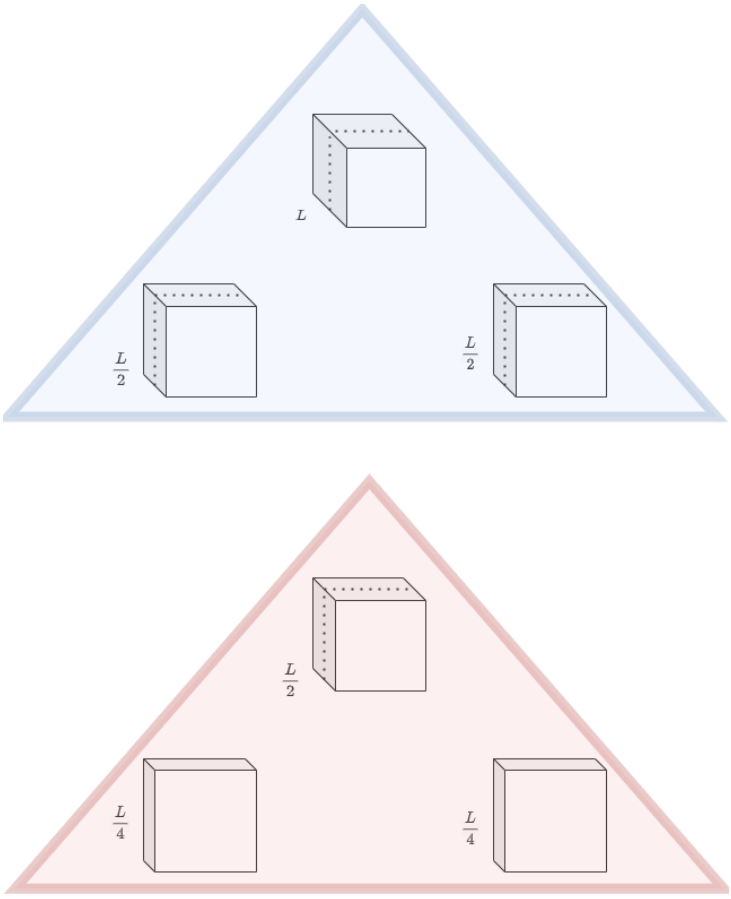


Figura 3.2: Triad of nodes from each node half-decomposition. Each one has a father, left child node and right child node.

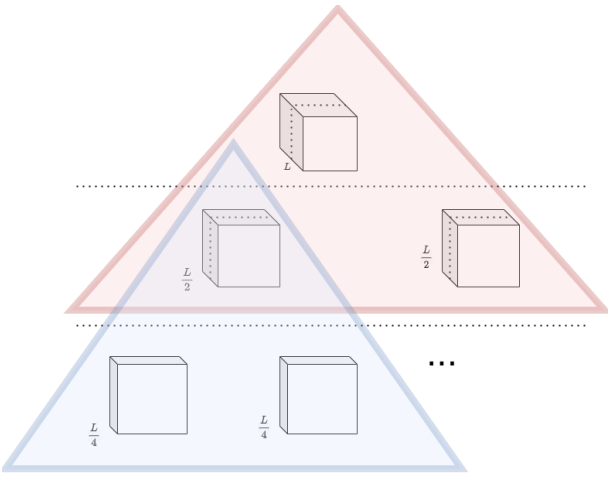


Figura 3.3: Triad of nodes from each node half-decomposition. Each one has a father, left child node and right child node.

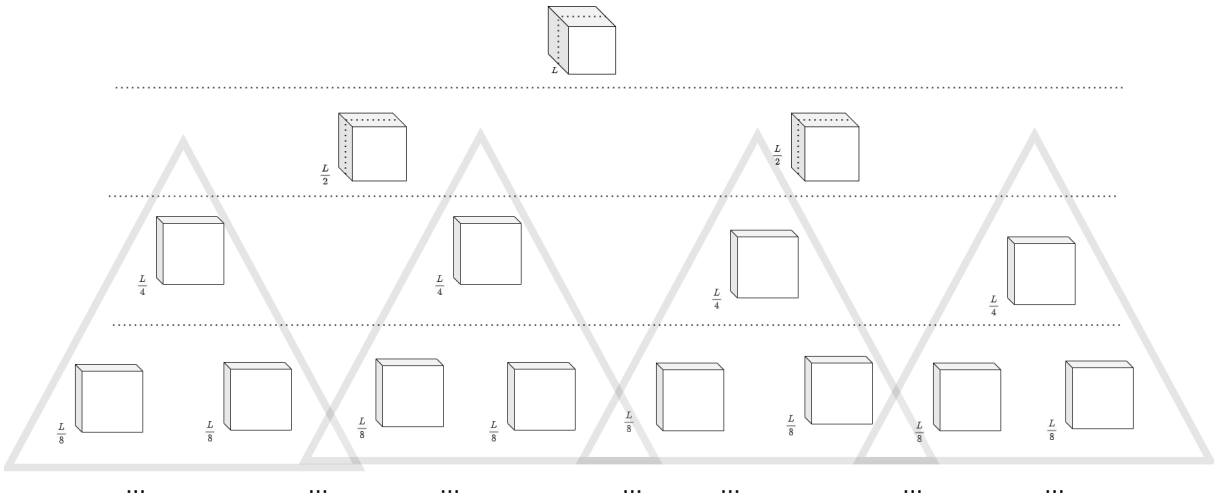


Figura 3.4: The decomposition tree divided in each 4 subtrees beginning from the 3rd level.

3.1.1 S3D-subtree(S3D-S)

The S3D-S algorithm marks the inaugural use of concurrency and thread-based programming within the Silhouette 3D (S3D) framework. This method hinges on partitioning the decomposition tree into distinct subtrees, as illustrated in Figure 3.5. The number of these independent subtrees directly correlates with the level at which the division is initiated. Therefore, selecting the level k (starting from 0) for subdivision results in 2^k subtrees, where k represents a non-zero natural number. In this approach, the autonomy of each subtree permits its independent encoding, enabling the assignment of each subtree to a distinct thread. This is achieved by initializing thread objects with the standard C++ thread library(`<threads>`) constructor. Within this constructor, the specific encoding function associated with a subtree is passed as a parameter, thus linking the function to the thread. After initializing thread objects for each subtree, the join method is invoked. This method ensures that the main program waits for all threads to complete their execution, thereby synchronizing the termination of all encoding processes. Considering the concurrent execution flows, the primary race condition observed involves the counting of the created threads and the identification of each subtree, which is necessary for segmenting the associated point cloud section. To address this, the increment in the number of executing threads is managed within a critical section. This section is safeguarded by calls to the mutex `lock` and `unlock` functions, ensuring that these operations are performed atomically, thereby preventing inconsistencies and ensuring the correct association of threads to their respective subtrees. In the context of a computer capable of managing hundreds of threads, this approach enables the concurrent encoding of the point cloud across n threads. Ensuring each subtree is encoded while maintaining the sequence of the

subtrees enables the original point cloud to be reconstructed during decoding by piecing together each segment.

As depicted, the encoding process initiates at the root node of each subtree, applying the S3D algorithm independently to each segment. Following the completion of all encodings, the bitstreams are consolidated in a sequential manner from left to right.

The decoding mirrors the encoding strategy, leveraging information provided in the header about the number of subtrees and the size of each bitstream. The decoder retrieves each bitstream, dedicating a thread to the decoding of each subtree using the same S3D mechanism. In the same way, the race condition occurs on the counting of the created threads, and identification of the associated point cloud segment. Once the point cloud of each subtree is reconstructed and the sequence within the tree is established, the point cloud segments are accurately aligned and merged, thereby reconstituting the original point cloud. The process of simultaneously decoding point cloud segments, reconstructing these segments, and merging them into a complete point cloud involves access to common variables and resources. Consequently, this concurrency introduces a race condition. To manage this issue effectively, mutex `lock` and `unlock` operations are employed around the procedures of retrieving the point cloud segment and integrating it with the final reconstructed point cloud. These operations ensure serialized access to shared resources, thereby preventing data corruption and ensuring consistency in the final output.

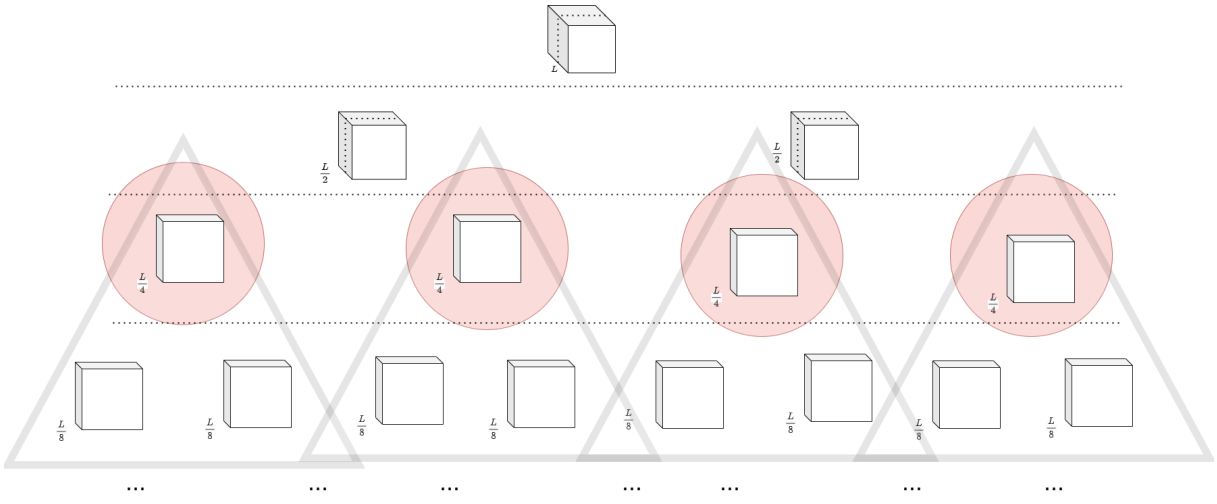


Figura 3.5: Decomposition tree of the S3D-S algorithm - highlighting the root node for each subtree.

3.1.2 S3D-subtree + Top-tree (S3D-ST)

In the *S3D-S* we observe that for each subtree encoded the root node is totally sent in the bitstream considering that it follows the S3D algorithm. If we have more subtrees,

more silhouettes of root node we are sending in the bitstream, making the encoding not much performative and advantageous in its compression rate loss compared with its speed gain. In order to reduce this compression rate loss for each root node subtree, the idea is not encode these nodes entirely, but find an alternative way of encode these nodes leveraging the decomposition from above layers.

In this way, the alternative version of the *S3D-S* proposes a similar idea, however the encoding adds an additional subtree: the top subtree. Thus, the threads objects creation, execution and race condition resolution are exactly as the *S3D-S* method, but now adding the top subtree thread. This approach is illustrated at image 3.6. Basically the top subtree has the responsibility of encode the root nodes from the children subtrees, thus, the root nodes is transmitted in the bitstream in a more performative way without losing its information. The top tree follows a very similar *S3D* encoding mechanism, however the *single mode* has its particularity. Instead of the *single mode* operation decompose point cloud slice in multiples unitary silhouettes slices, the *single mode* decomposes until reach the silhouettes from blocks that belongs to the level n chosen to the subtrees decomposition. This happens because theoretically the subtree is limited to the top subtree division, not being allowed to surpass the levels below where the subtrees is already encoded. Otherwise, redundant data would be sent along with the subtrees encodings. The subtrees encoding follows the same mechanism as the *S3D-S*.

In the decoding the same idea is repeated. First, the top tree is decoded, then the root nodes from the subtrees is restored. With these nodes restored, we can compute the decoding on each subtree following the same idea. It is important to note that once there is an additional step involving the top subtree encoding the root nodes from each subtree, we would require n threads plus the top tree thread. Thus, we have $n + 1$ threads used in the encoding process.

3.2 S3D Block Mode

Prior to the development of the *block mode* concept, a series of tests and performance evaluations were conducted on the C++ implementation of S3D, focusing on dyadic decomposition and single mode testing. The results indicated that the single mode often outperformed or matched the efficiency of dyadic decomposition encoding in most scenarios. Consequently, it became apparent that single mode could potentially offer robust performance on its own, eliminating the need for dyadic decomposition testing as initially conceived in the original framework.

Through this analysis, two principal advantages emerged from favoring the single mode approach. The first advantage is the simplification of the encoding process. Unlike the

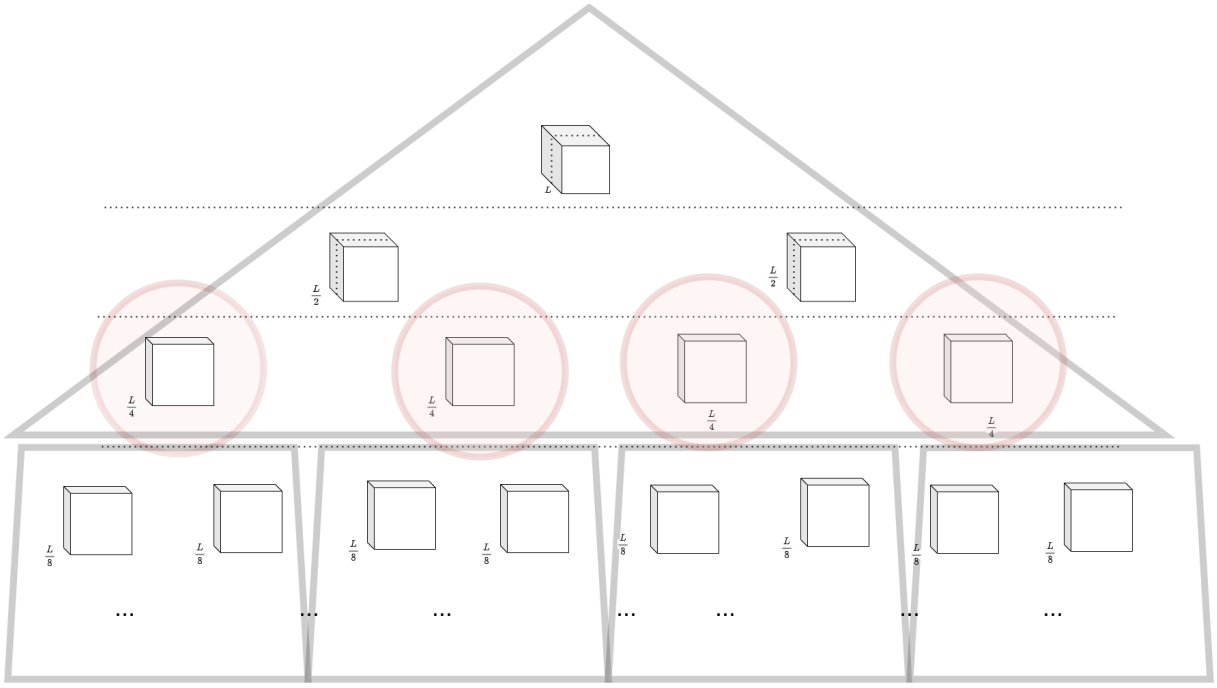


Figura 3.6: Decomposition tree of the S3D-ST algorithm - highlighting the root node for each subtree. Here, we can observe that the encoding of the subtrees and the root node are in different threads

comprehensive decomposition required by recursive binary division, the point cloud is directly segmented into unitary silhouette slices. This direct approach not only simplifies the encoding procedure but also significantly accelerates it, as it demands less computational effort.

The second advantage, though indirectly mentioned, pertains to the increased efficiency and speed of the encoding process as a direct result of this simplification. With fewer steps involved in breaking down the point cloud, the encoding becomes more straightforward and fast. However, it is important to note that this efficiency comes with a trade-off: The single mode is more direct method results in a loss of contextual depth. Since fewer silhouettes are encoded, the algorithm has less contextual data to leverage during the pixel encoding process, a domain where the dyadic decomposition technique offers superior performance by enriching the context through its more layered approach.

In seeking an alternative to dyadic decomposition that retains context while maintaining efficiency, the investigation led to a more adaptable decomposition strategy. Instead of solely bifurcating the point cloud, considering its division into powers of two offered a promising foundation for the block mode strategy, thereby inspiring one of the novel algorithms introduced in this study.

3.2.1 Standard (S3D-BM)

The standard *block mode* algorithm simplifies the point cloud encoding process into two primary steps. The initial step involves dividing the original point cloud into m blocks, where m is a power of two, represented as 2^k, with k being a non-zero natural number. This division strategy becomes clearer when examining the decomposition tree, as shown in Figure 3.7. Similar to the original S3D algorithm, the point cloud is initially split in halves. However, the *block mode* diverges by directly slicing the point cloud into blocks at the kth level of decomposition, as illustrated for $k = 2$. Consequently, the m blocks produced correspond to 2^k, aligning with the level k where the division occurs. This method effectively jumps directly to the level k in the decomposition tree, bypassing intermediate steps. The encoding process begins with the root node, followed by encoding each silhouette of the block, which then serves as context for the CABAC encoding.

After this initial decomposition into blocks, each block undergoes a single mode decomposition as detailed in Figure 3.7. This results in the encoding of each point cloud slice within the blocks, culminating in the final bitstream. The decoding process mirrors the encoding steps, ensuring the restoration of the original point cloud. This approach is notably more straightforward and direct compared to the exhaustive performance testing of methods within the original S3D algorithm, suggesting a potential for faster performance due to its reduced computational complexity.

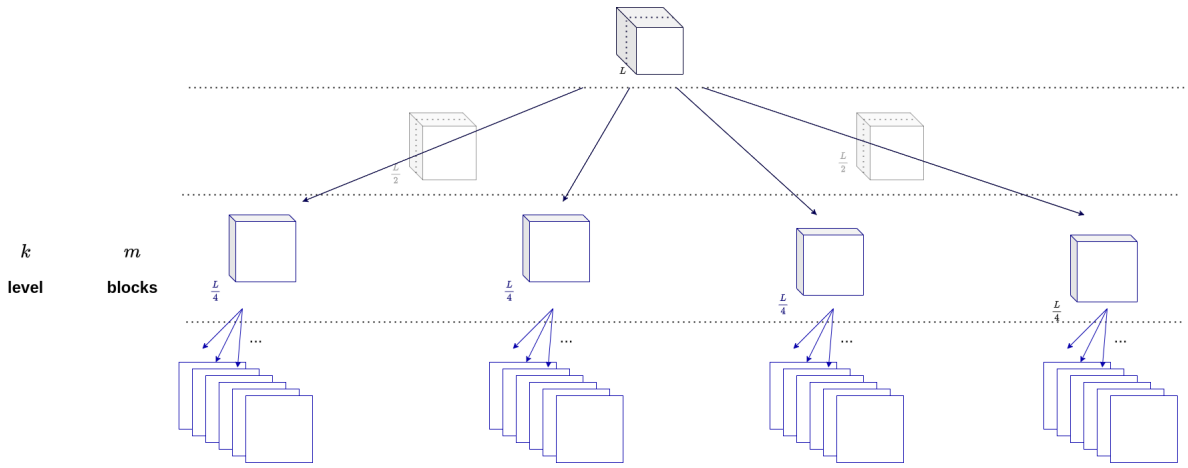


Figura 3.7: Decomposition tree of the S3D-standard-block-mode algorithm - in the case of this example, $k = 2$ and $m = 4$

However, while this method streamlines the S3D algorithm, it also highlights a need for enhanced versatility, particularly in real-time acquisition scenarios. For instance, a camera capturing a point cloud progressively along a given axis may find this method unsuitable, as it relies on prior knowledge of the entire point cloud geometry. This limitation is significant in real-time applications where geometry is captured incrementally. To address

this challenge, an *Inverted mode* of the Block mode algorithm is introduced, aiming to adapt the encoding process to scenarios where point clouds are progressively acquired, thereby enhancing the applicability of the method in dynamic environments.

3.2.2 Inverted Mode (S3D-IM)

The *inverted mode* is essentially a reversal of the procedural steps outlined in the standard *Block Mode* algorithm, while still retaining the foundational elements of each phase—namely, executing the single mode on unitary silhouettes followed by the encoding of silhouette blocks. Initially, the inverted mode begins with the encoding of unitary silhouettes from the point cloud utilizing the single mode approach, as illustrated in Figure 3.8. This methodology aligns well with scenarios where the point cloud geometry is progressively captured over time.

Once the geometry of the point cloud is fully captured within a defined volumetric space, the process progresses to the next phase. The task of deducing the blocks from the point cloud becomes straightforward by amalgamating the unitary silhouettes into widths that correspond to the desired block dimensions, adhering to the formula $width = L/2^k$, where L represents the total width of the point cloud and k denotes the level of decomposition. This procedure is depicted in Figure 3.8. Upon determining all blocks, the silhouettes of each block are encoded, culminating in the encoding of the root node. The decoding sequence mirrors that of the *Block Mode* strategy, albeit in reverse, meticulously following the steps delineated above.

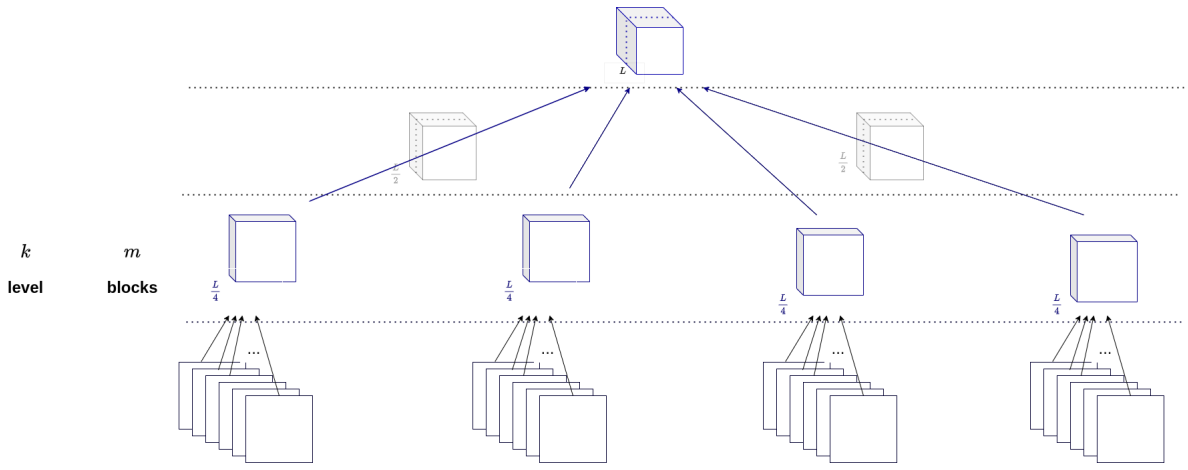


Figura 3.8: Decomposition tree of the S3D-inverted-mode algorithm - in the case of this example, $k = 2$ and $m = 4$

Comparing the two methodologies, it becomes apparent when examining a representation of the decomposition tree for the block mode, as depicted in Figure 3.9, that the

standard and the inverted methods fundamentally share the same structure, differing primarily in their traversal approach.

The standard mode adopts a *top-down* methodology, commencing from the root and progressing downwards through the decomposition tree. Conversely, the inverted mode employs a *bottom-up* strategy, initiating from the unitary silhouettes and working upwards towards the root. Despite both methods relying on identical computational bases, they are not expected to yield equivalent encoding performance. This disparity arises from the differing nature of context information available to each method. The standard approach benefits from a richer context in the initial pixels encoded, offering a more detailed basis for subsequent encoding steps. On the other hand, the inverted mode starts with pixels at the edge slices of the point cloud, which generally provide less valuable context for the overall encoding of the point cloud.

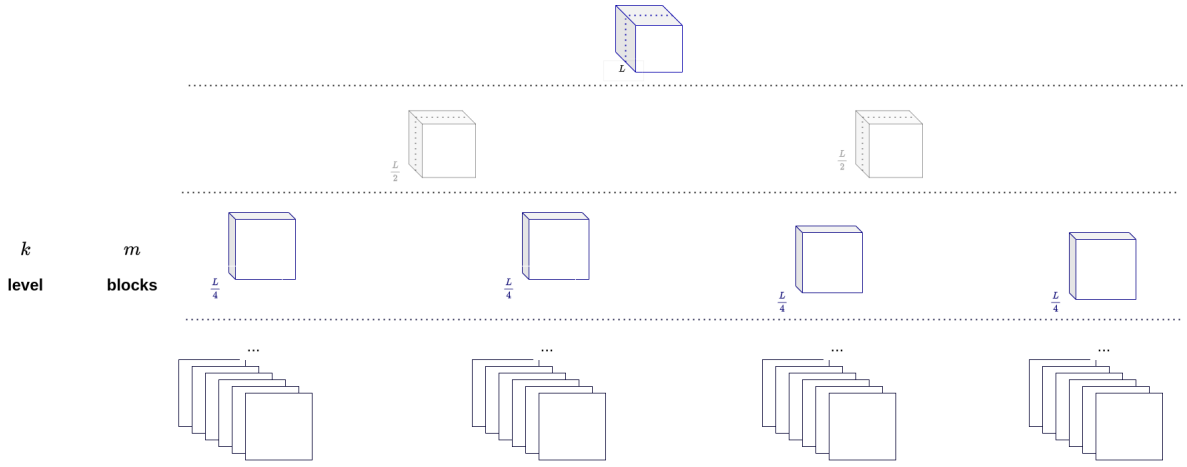


Figura 3.9: The general decomposition tree of the S3D block mode algorithm. Here are clear the basic structure of the algorithm, and how it can branch in the different algorithms in figures 3.7 and 3.8, changing the tree traversing (top-down or bottom-up).

3.3 Unified header

All four proposed algorithms, along with the original S3D algorithm, were integrated as modes of the proposed codec, avaible in the repository of Github [19]. This integration allows for the selection of one of the five S3D variations during the encoding process. Depending on the selected algorithm and the corresponding user-specified parameters, all relevant data are collected and inserted into a universal header, which is common across all S3D variation bitstreams. The structure of this header is illustrated in Figure 3.10.

Given this context, the header can be described as follows:

1. **nBits** - Specifies the resolution of the point cloud.

2. **axis** - Indicates the axis chosen for point cloud encoding.
    - 1 corresponds to the X-axis.
    - 2 corresponds to the Y-axis.
    - 3 corresponds to the Z-axis.
3. **algorithmChoice** - Denotes the algorithm selected for encoding.
4. **m** - Represents the bit length of the CABAC tag.
5. **nc1D** - The number of contexts used in 1D for silhouette CABAC encoding.
6. **nc2D** - The number of contexts used in 2D for silhouette CABAC encoding.
7. **nc3D** - The number of contexts used in 3D for silhouette CABAC encoding.
8. **k** - Relevant to the S3D Block Mode and S3D Inverted Mode algorithms. Param that indicates the number of levels traversed from the root to reach the level where block mode encoding is applied.
9. **nParallelism** - Pertains to the S3D-S and S3D-ST algorithms. It is the logarithm base 2 of the number of threads used during encoding.
10. **lengthBitstreamParam** - Refers to the size of the bitstream generated during the silhouettes tree encoding process.

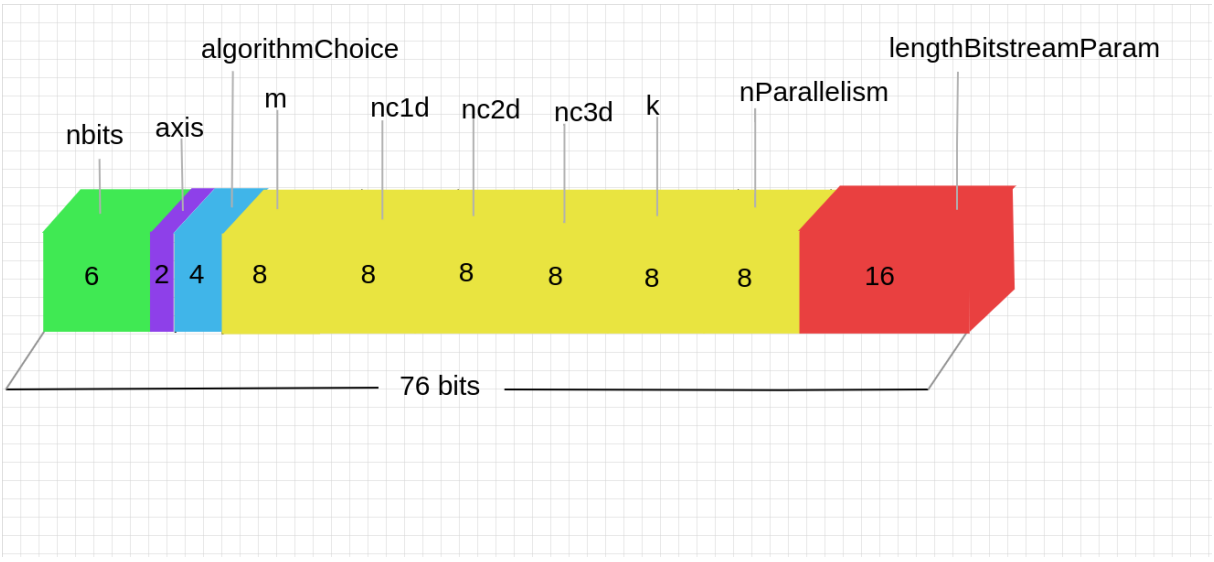


Figura 3.10: Unified header structure illustration, relating fields and its bit length.

It is important to note that depending on the algorithm selected, certain fields may not be relevant for decoding and are therefore ignored. Nonetheless, these fields are still populated with zero values across all corresponding bits.

3.4 Arithmetic Coding Contexts

In the original silhouette coding, the 3D context choices, as explained in 2.4.2, follow the order shown in figure 3.11a. As we can observe, this particular selection was not based on any specific motivation but rather on the author's convenience and personal preference. In this work, however, a new sequence has been chosen, as illustrated in figure 3.11b. This new arrangement was made to improve the readability of the pixel context indexing in the code implementation, as seen when comparing the old version (listing 3.1) to the new version (listing 3.2).

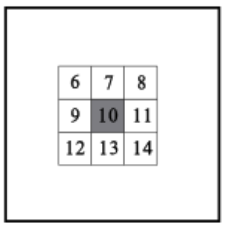
Listing 3.1: Previous version 3D contexts

```
pixels[0] = image.PixelPresent(y - 1, x - 1);
pixels[1] = image.PixelPresent(y - 1, x);
pixels[2] = image.PixelPresent(y - 1, x + 1);
pixels[3] = image.PixelPresent(y, x - 1);
pixels[4] = image.PixelPresent(y, x);
pixels[5] = image.PixelPresent(y, x + 1);
pixels[6] = image.PixelPresent(y + 1, x - 1);
pixels[7] = image.PixelPresent(y + 1, x);
pixels[8] = image.PixelPresent(y + 1, x + 1);
```

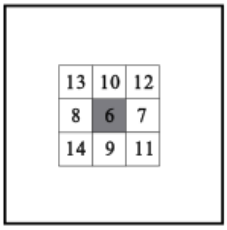
Listing 3.2: New version 3D contexts

```
pixels[0] = image.PixelPresent(y, x);
pixels[1] = image.PixelPresent(y, x + 1);
pixels[2] = image.PixelPresent(y, x - 1);
pixels[3] = image.PixelPresent(y + 1, x);
pixels[4] = image.PixelPresent(y - 1, x);
pixels[5] = image.PixelPresent(y + 1, x + 1);
pixels[6] = image.PixelPresent(y - 1, x + 1);
pixels[7] = image.PixelPresent(y - 1, x - 1);
pixels[8] = image.PixelPresent(y + 1, x - 1);
```

Figura 3.11: 3D pixels contexts sequence for the previous and new version of the S3D.



(a) Previous S3D version 3D contexts choices.



(b) New S3D version 3D contexts choices.

Capítulo 4

Results and analysis

With an in-depth exploration of the mechanisms behind the algorithms presented and proposed in this study, we now shift our focus to evaluating their performance. This evaluation will not only detail each performance metrics of the algorithm, but also elucidate their distinctive features through study cases and various performance assessments involving different parameter settings. The results of this analysis will allow us to more thoroughly examine and verify the hypotheses posited in the section of this work dedicated to the proposed algorithms.

First, the performance of the new S3D C++ implementation rate was assessed, highlighting its performance difference with the previous Matlab version. For this, different combinations of 2D and 3D contexts were tested over the dataset before, in order to determine which amount of contexts would give the best rate. With this combination, a definitive version of the standard S3D C++ is obtained, which allows one to obtain the definitive results of the S3D final version.

The rate and time performance of the thread-based algorithms S3D-S and S3D-ST are evaluated using different numbers of threads. Each thread is applied to encode the point cloud as independent subtrees. The objective is to analyze the relationship between the number of threads and the performance in time and rate of the algorithm. Additionally, the rate performance of the S3D Block Mode and Inverted Mode is examined. This new approach explores the concept of jumping directly to specific levels of dyadic decomposition, offering a simpler and more direct method. Moreover, this dynamic allows the algorithm to be applied in previously unexplored contexts, such as *real-time acquisition* encoding. By analyzing its behavior under different parameters and approaches, top-down (block mode) and bottom-up (inverted mode), this study aims to explore the potential of the S3D algorithm for various applications and identify its most efficient configuration.

All rate and time results were collected using the Microsoft Voxelized Upper Bodies [14] and 8i Voxelized Full Bodies (8iVFB v2) [20] as dataset. All results were collected on

a computer with an Intel® Core™ i7-5820K CPU @ 3.30GHz × 12 running on Ubuntu 22.04 LTS, compiling the program using C++17. It is important to note that each codec can be applied along any of the available axes of a point cloud: x, y, and z. Encoding is performed along each axis with the optimal rate selected from them.

4.1 2D and 3D contexts setting

Before assessing the S3D rate performance, an important aspect that was reviewed and explored was the combination of 2D and 3D context amounts. From a range of different combinations, each one were tested to determine which combination yields the best rate performance, considering the primary goal of the codec(rate performance). For this, the last three frames of each point cloud in the *upper bodies* and *full bodies* datasets were encoded, with the 2D context ranging between $[4, 10]$ and the 3D context varying between $\{1, 5, 9\}$.

For each combination of 2D and 3D contexts (e.g., (4,1) represents all frames encoded using four 2D contexts and one 3D context, (4,5) uses four 2D contexts and five 3D contexts, and so on), the rate and time were collected. The average and standard deviation of these values were computed.

These results were plotted in a rate vs. time (bits per occupied voxel vs. seconds) scatter distribution, with the average and standard deviation as illustrated in figure 4.1. It was observed that the best results in terms of rate were obtained with a 3D context amount equal to 9. To refine the selection, a second scatter plot, focusing only on the 3D context equal to 9, was produced in figure 4.2. This scatter plot revealed more details, showing that a configuration with eight 2D contexts offers the best rate performance.

Further insights can be found in table 4.1, where rate is in *bits per occupied voxel* and time in *seconds*. Thus, the optimal configuration for S3D performance is achieved when eight 2D contexts and nine 3D contexts are applied in the silhouette encoding. All the following results in this work will be based in this configuration of contexts.

As discussed in Section 3.4, the selection of 2D and 3D pixel contexts for silhouette encoding was updated in the new implementation. In the previous version, the 2D and 3D contexts, illustrated in Figure 4.3, consisted of five 2D contexts and nine 3D contexts. In the updated version, now employing eight 2D contexts and nine 3D contexts, the encoding process is represented in Figure 4.4.

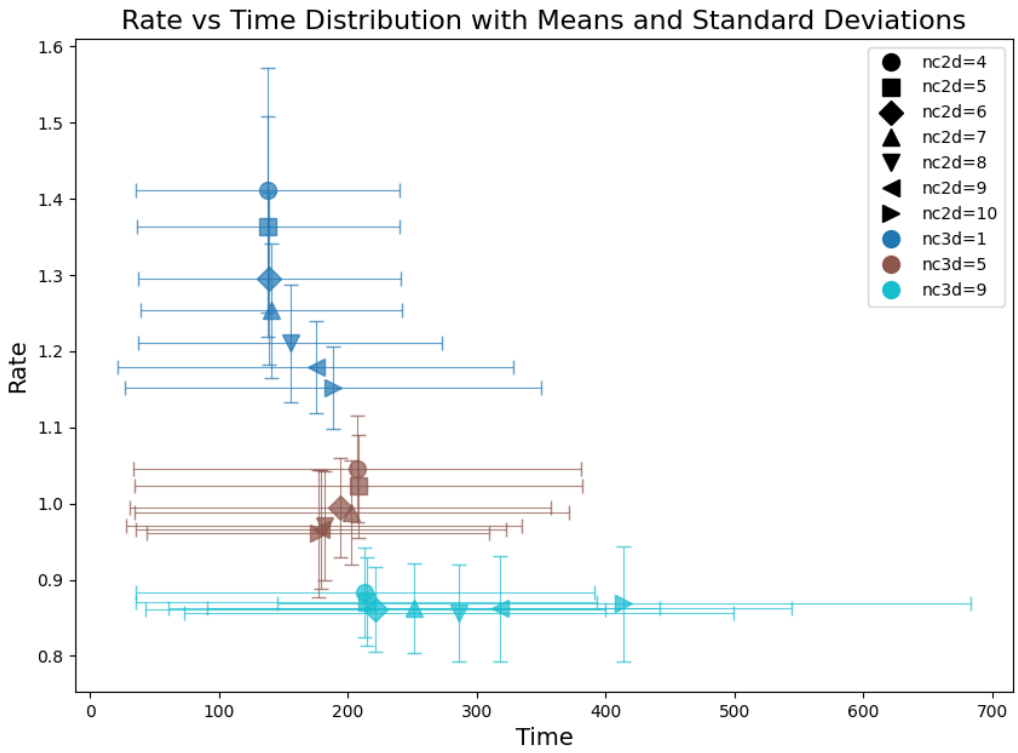


Figura 4.1: S3D standard contexts combination scatter plot on rate x time distribution, bits per voxel x seconds

4.2 Standard S3D C++ algorithm results

Building on the enhancements made to the C++ implementation of the S3D algorithm—specifically the integration of single-mode encoding and extensive testing across the decomposition tree—it is crucial to document the key outcomes of this work as discussed in Section 1.4. Additionally, as detailed in Section 2.4.2, the objective of the testing process is to determine the most efficient approach at each node within the decomposition tree, whether through dyadic decomposition or single-mode encoding. This strategy enables a systematic identification of performance-optimized decomposition pathways on a global scale. Moreover, as discussed in the previous section, the optimization of the number and order of 2D and 3D contexts has led to significant rate performance improvements. This will be assessed in the rate performance results presented in this section.

All results across the datasets were obtained by averaging the first 100 frames. As illustrated in Table 4.2, introducing an additional encoding option within the decomposition tree - namely, the single mode - enables the exploration of alternative point cloud encoding strategies beyond dyadic decomposition. When evaluating the performance on the Microsoft Voxelized 9-bit Upper Bodies database, the enhanced S3D C++ imple-

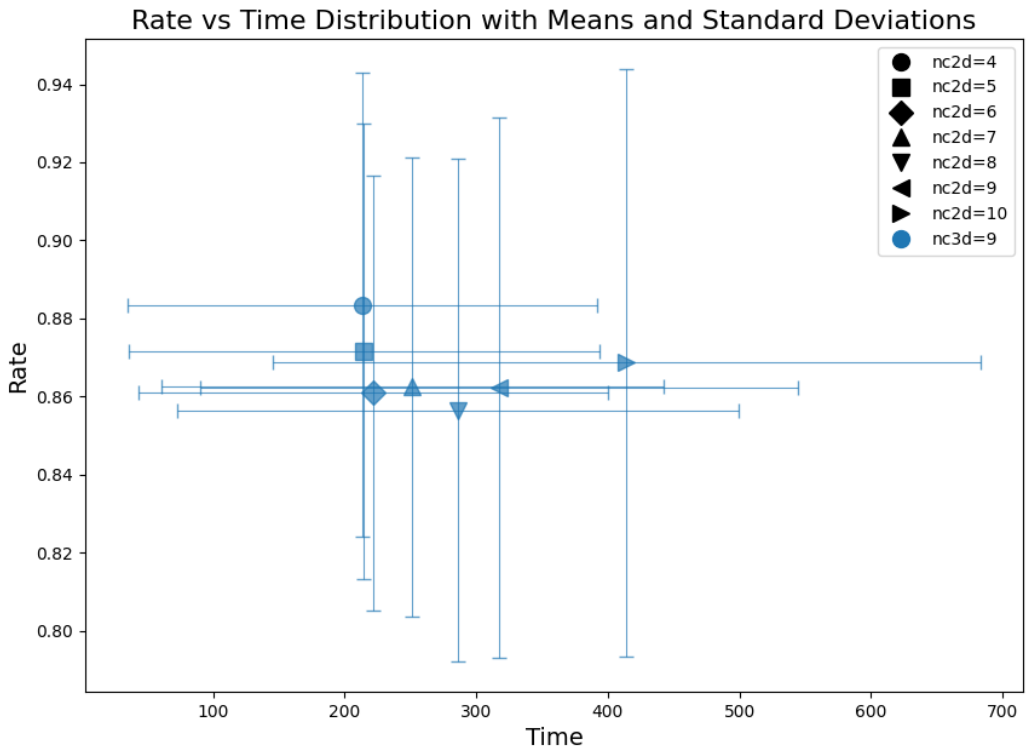


Figura 4.2: S3D standard contexts combination scatter plot for 3D contexts equal to 9, bits per occupied voxel vs seconds

mentation demonstrates a notable outperformance of 5.26% in compression rate average efficiency relative to the best result achieved by other state-of-the-art algorithms, the CS-S4D. Besides, when compared to original S3D Matlab implementation it presents a relative outperformance of 17.43%.

Furthermore, while the S3D C++ implementation may not surpass the performance of the TMC13 algorithm on the 8i Voxelized Full Bodies database, it nevertheless outperforms the majority of competing algorithms, having a relative outperformance of 2.41% over the CS-S4D. The inferior performance compared to TMC13 primarily results from the transmission of excessive empty voxel data in the bitstream when applied to 10-bit resolution point clouds. This leads to significant redundancy of blank volumes that the S3D strategy fails to address, a limitation that becomes particularly pronounced with high-resolution point clouds.

In addition, it achieves a relative outperformance of 15.3% of the average rate over the original S3D algorithm. Notably, its performance remains competitively close to that of the TMC13 algorithm, despite the S3D's reliance on a more constrained set of encoding options — limited to just two choices. This underscores the effectiveness of the proposed

nc2d	nc3d	mean rate(bpov)	deviation rate(bpov)	mean time(s)	deviation time(s)
8	9	0.856401	0.064370	286.103296	213.281962
6	9	0.861040	0.055692	221.681815	178.453861
9	9	0.862302	0.069131	317.667704	227.129691
7	9	0.862542	0.058770	251.287926	190.878672
10	9	0.868723	0.075311	414.098185	269.033280
5	9	0.871494	0.058336	214.422111	178.844944
4	9	0.883520	0.059436	213.411111	178.216343
10	5	0.960845	0.083371	176.938852	132.526578
9	5	0.966207	0.077405	179.115852	143.921869
8	5	0.970450	0.071464	181.820519	153.356921
7	5	0.988746	0.068081	203.055296	168.247118
6	5	0.995178	0.065271	194.302778	163.187633
5	5	1.022610	0.066861	208.079259	173.735663
4	5	1.045876	0.069439	207.292519	174.123963
10	1	1.152158	0.053405	188.543444	161.173013
9	1	1.178870	0.060332	175.178778	153.541889
8	1	1.210307	0.076666	155.263074	117.542732
7	1	1.253442	0.088000	140.352593	101.424913
6	1	1.295345	0.113604	139.082296	101.616206
5	1	1.363893	0.144390	138.088296	101.875288
4	1	1.410650	0.160503	138.085407	102.309323

Tabela 4.1: Results of rate (bits per occupied voxel) and time (seconds) from standard S3D encoding based on different amount of 2D and 3D contexts, sorted in descending order of mean_rate

S3D enhancements in achieving superior compression rates, even within the context of limited encoding strategies compared to TMC13.

4.3 S3D-S and S3D-ST results

From the results of migrating the classic S3D to C++, we have observed a significant performance improvement. Building upon this version, the proposed multi-threaded variants, S3D-S and S3D-ST, will be evaluated in terms of two crucial aspects: time and compression rate. It is important to highlight that the primary goal of both algorithms is to improve the execution time of S3D, addressing its previously lengthy run times.

However, this improvement in time efficiency comes at the cost of compression performance, particularly due to the separation of CABAC contexts among different subtrees. In our rate analysis, we will explore the relationship between improvements in running time and the associated trade-offs in rate performance. Also, all results across the datasets were obtained by averaging the first 12 frames, one sample per frame.

Sequence	TMC13 v19 [21]	FRL [22]	S3D Matlab [23]	CS-S4D [24]	S3D C++
Andrew	1.03	1.00	1.12	**0.95**	**0.95**
David	0.95	0.94	1.06	0.94	**0.87**
Phil	1.05	1.02	1.14	1.02	**0.95**
Ricardo	0.97	0.93	1.04	0.90	**0.87**
Sarah	0.96	0.95	1.07	0.92	**0.88**
Average	0.99	0.97	1.09	0.95	**0.90**
LongDress	**0.76**	0.86	0.95	0.88	0.80
Loot	**0.71**	0.83	0.92	0.84	0.76
RedAndBlack	**0.84**	0.94	1.02	0.94	0.87
Soldier	0.76	0.88	0.96	**0.65**	0.81
Average	**0.77**	0.88	0.96	0.83	0.81

Tabela 4.2: S3D standard comparison with state-of-the-art codecs for the first 100 frames average shown in bits per occupied voxel (bpov).

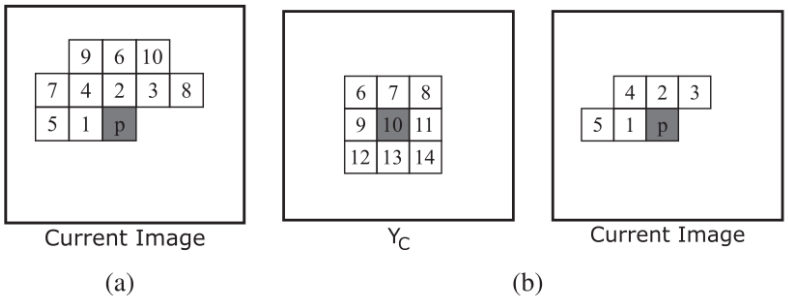


Figura 4.3: Previous version of S3D utilizes pixel contexts, with five 2D contexts and nine 3D contexts. In the case of encoding only with 2D contexts, the selected contexts are represented in (a). When 3D contexts are used, the selected contexts are represented in (b).

4.3.1 Running time

Considering the primary objective of the thread-based algorithms *S3D-S* and *S3D-ST* — to reduce encoding time by segmenting the encoding process into concurrent execution flows, represented by each subtree — the initial results focus on the relationship between the encoding time of each algorithm and the number of threads utilized.

Given that each subtree within the described algorithms functions as an independent unit of encoding, carried out by a separate thread, and thus partitions the computation of the encoding bitstream across the exact number of subtrees generated, it is anticipated that the more the decomposition tree is segmented into subtrees, the less global computation time is required. This expectation hinges on the optimized utilization of the CPU facilitated by the thread-based approach. These considerations led us to analyze the chart in Figure 4.5, which delineates the correlation between the execution time of

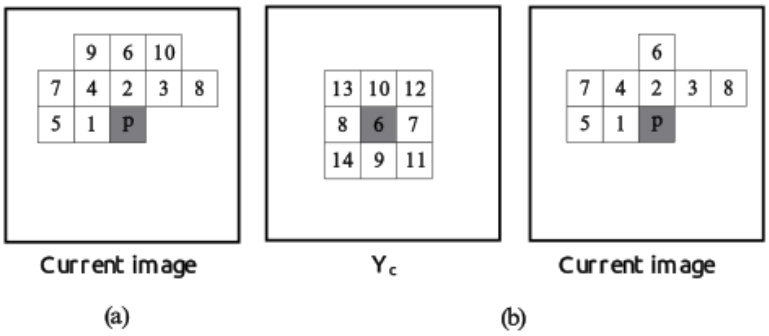


Figura 4.4: New version of S3D utilizes pixel contexts, with eight 2D contexts and nine 3D contexts. In the case of encoding only with 2D contexts, the selected contexts are represented in (a). When 3D contexts are used, the selected contexts are represented in (b).

each algorithm and the number of threads deployed in the computation. In this plot, the *upper bodies* time performance is result of averaging all datasets results in seconds, for each number of threads. The same is applied to *upper bodies*.

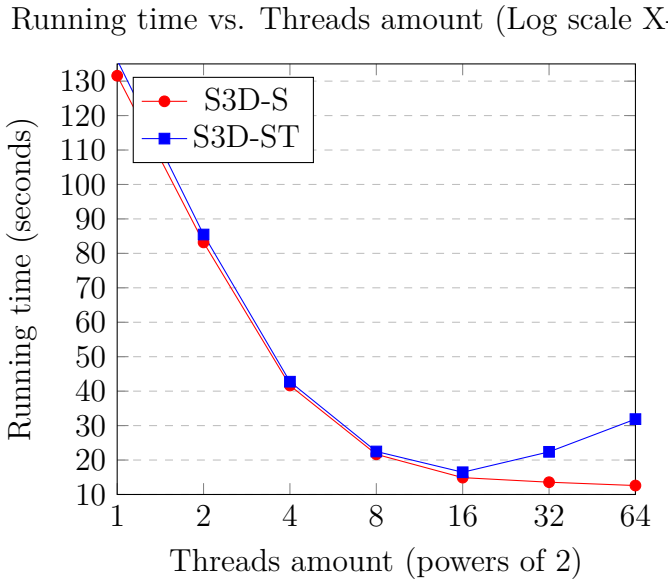


Figura 4.5: Average Running Time for the first 12 frames from each dataset over different amount of threads for S3D-S and S3D-ST.

Observations reveal that the utilization of additional threads — corresponding to a greater subdivision of the decomposition tree into subtrees — results in reduced time to produce the final bitstream, which amalgamates the bitstreams generated by each subtree. Specifically, applying a decomposition into 4 subtrees, as illustrated in the chart, leads to a reduction of approximately 60% in time compared to employing a single thread,

synonymous with the standard S3D algorithm. Thus, it is evident that segmenting the encoding process of the algorithm significantly decreases its operational duration.

Another notable finding is the performance parity between *S3D-S* and *S3D-ST* up to 16 threads. Beyond this point, the *S3D-S* algorithm demonstrates superior performance over *S3D-ST*. A plausible rationale for this divergence is the extra computational overhead introduced by the top subtree encoding in *S3D-ST*, as opposed to the singular encoding task of one silhouette from the parent node for each subtree in *S3D-S*. It is critical to acknowledge that this additional computation represents a deliberate compromise between time expenditure and compression efficiency. The purpose behind this compromise — to enhance the encoding of silhouette pixels through the provision of additional context — will be further elucidated in the forthcoming analysis concerning the interplay between compression rate and the quantity of threads employed.

4.3.2 Rate

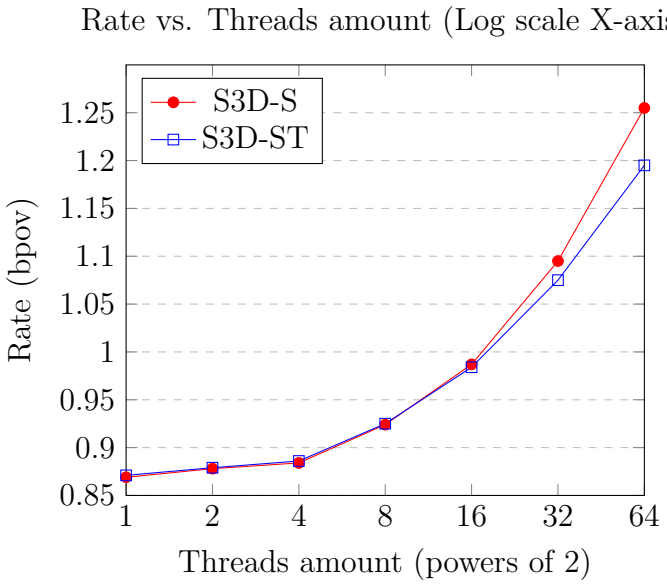


Figura 4.6: Average Rate for the first 12 frames from each dataset over different amount of threads for S3D-S and S3D-ST.

With the preliminary analysis focused on the correlation between running time and the number of threads, we now shift our attention to the influence of thread quantity on the ultimate compression rate. This investigation yields two critical insights: firstly, the manner and extent to which thread count impacts algorithm performance; and secondly, and more crucially, the trade-off between running time and compression rate, both of which are contingent on the amount of threads utilized. As the number of threads escalates, a worsen compression rate is anticipated due to the loss of contextual

information during the encoding of silhouettes for each subtree, attributed to the reduced number of symbols encoded by each codec of the subtree. Consequently, we expect an direct relationship between compression rate and running time - adjustments in thread quantity within the same dataset will expedite algorithm execution, albeit at the expense of compression efficiency.

This result illuminates the potential to discern an optimal thread usage value, indicative of an encoding that achieves the best compromise between compression rate and time efficiency. This optimum not only signifies the peak relative performance of the algorithm, but also serves as a benchmark for application-specific performance calibration. For instance, in scenarios necessitating swift compression, such as *on-the-fly* applications, a slight compromise on compression quality for enhanced time performance may be deemed acceptable. Conversely, if superior compression quality is desired without a significant detriment to time efficiency, this reference value guides the adjustment of thread count to achieve the preferred balance. In contexts where time constraints are negligible, reverting to a single-thread approach, akin to the original S3D algorithm, might prove optimal. This flexibility and the range of choices offered by the multi-threaded S3D algorithms underscore their significant contribution to the realm of geometric point cloud encoding.

Referencing the chart in Figure 4.6, the anticipated direct correlation between compression rate and thread count is evident. In this plot, all results is obtained as the running time chart in figure 4.5, by averaging the results across the datasets. Notably, up to 16 threads, the performance of both algorithms is similar across datasets (Upper Bodies and Full Bodies); beyond this threshold, as hypothesized, the *S3D-ST* demonstrates superior efficiency, attributable to the additional computational load imposed by encoding the top subtree.

This extra computational effort, while increasing runtime, results in more efficient encoding compared to the *S3D-S* approach of transmitting the complete image for each subtree. In the *S3D-ST* model, encoding is confined to the root node of the entire decomposition tree and the subsequent top subtree, enhancing context aggregation and reducing the bit requirement for conveying each parent of the subtree silhouette.

In Table 4.3 and 4.4, the comprehensive performance metrics of rate and running time from each algorithm are presented for the point clouds of each dataset. It is noteworthy that employing merely a single thread across these algorithms yields results akin to those achieved through direct application of the S3D algorithm.

Finally, an interesting analysis arises when both the rate and running time are plotted together. As shown in Figure 4.7 and Figure 4.8, increasing the number of threads leads to a higher rate and a reduction in running time. Conversely, when reduced, lower rate,

Tabela 4.3: Rate results using the first 12 frames average of each sequence. All rates are in bits per occupied voxel (bpov).

		S3D	S3D-S						S3D-ST					
Database	Sequence	1	2	4	8	16	32	64	2	4	8	16	32	64
Microsoft Voxelized Upper Bodies [25]	Andrew	0.948	0.981	1.008	1.048	1.105	1.209	1.359	0.987	1.016	1.052	1.106	1.198	1.317
	David	0.922	0.943	0.969	1.026	1.109	1.212	1.367	0.950	0.981	1.036	1.110	1.199	1.321
	Phil	0.974	0.998	1.028	1.068	1.143	1.257	1.418	1.002	1.034	0.965	1.143	1.243	1.369
	Ricardo	0.869	0.878	0.884	0.924	0.987	1.095	1.255	0.878	0.886	0.925	0.984	1.075	1.195
	Sarah	0.925	0.958	0.974	1.012	1.074	1.166	1.314	0.966	0.978	1.020	1.078	1.158	1.273
	Average	0.948	0.981	1.008	1.048	1.105	1.209	1.359	0.987	1.016	1.052	1.106	1.198	1.317
8i Voxelized Full Bodies [20]	LongDress	0.791	0.802	0.817	0.840	0.864	0.913	0.997	0.810	0.824	0.846	0.871	0.914	0.981
	Loot	0.751	0.752	0.767	0.782	0.813	0.866	0.954	0.761	0.777	0.791	0.821	0.865	0.936
	RedAndBlack	0.857	0.866	0.871	0.892	0.927	0.982	1.077	0.879	0.884	0.909	0.939	0.990	1.065
	Soldier	0.805	0.806	0.822	0.835	0.869	0.925	1.023	0.805	0.822	0.835	0.867	0.916	0.995
	Average	0.798	0.807	0.819	0.837	0.868	0.974	1.013	0.854	0.853	0.871	0.926	0.974	1.017

Tabela 4.4: Running time results using the first 12 frames average of each sequence. All running time are in seconds (s).

		S3D	S3D-S						S3D-ST					
Database	Sequence	1	2	4	8	16	32	64	2	4	8	16	32	64
Microsoft Voxelized Upper Bodies [25]	Andrew	109,748	71,669	34,624	20,469	11,811	9,437	8,916	73,398	35,364	20,914	13,517	14,210	21,210
	David	125,801	79,701	37,956	17,950	14,891	13,566	12,598	81,508	38,908	19,264	16,455	22,381	31,880
	Phil	131,569	83,179	41,158	21,362	14,386	12,115	11,248	85,470	42,156	22,437	15,062	20,139	28,895
	Ricardo	85,172	48,394	38,120	18,970	9,745	6,859	6,314	51,087	39,216	19,394	9,695	9,550	14,644
	Sarah	121,234	75,143	41,631	21,658	13,231	10,912	10,165	76,825	42,739	22,483	13,748	17,416	25,298
	Average	114,705	71,617	38,698	20,082	12,813	10,578	9,848	73,465	39,917	21,498	14,171	14,837	20,385
8i Voxelized Full Bodies [20]	LongDress	496,048	239,225	113,646	75,865	38,961	25,724	21,241	241,607	157,631	72,764	39,841	29,813	43,994
	Loot	269,025	230,638	127,702	67,369	32,875	23,263	20,733	239,091	132,851	70,215	35,328	27,647	41,210
	RedAndBlack	454,626	218,969	143,152	67,454	32,163	21,699	18,392	204,987	146,288	70,016	32,543	22,977	34,744
	Soldier	629,549	313,226	167,726	89,125	41,551	28,547	24,719	325,348	174,935	93,721	46,512	37,958	55,504
	Average	462,312	250,515	138,057	74,953	36,388	24,808	21,271	247,007	153,681	77,730	38,057	29,349	43,866

however, at the cost of higher running time. Although this project does not define a specific optimal value, as previously mentioned, the point where the two curves intersect serves as a useful reference. This intersection suggests that a value near 8 threads may be an effective choice.

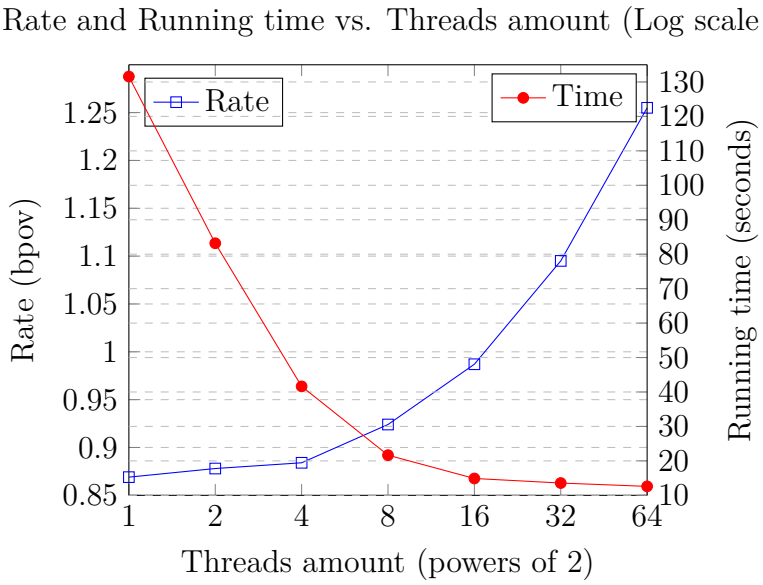


Figura 4.7: Rate and running time plot over different amount of threads for S3D-S.

Here, it is more clear the features and characteristics observed. The more threads are used, the less performative the algorithm becomes. Thus, when we prioritize the performance over running time, directly using S3D algorithms becomes more advantageous over the thread-based algorithms. Moreover, when we analyze the efficiency among the thread based algorithms, the *S3D-ST* performance, as noted before, is better than the *S3D-S* algorithm when the number of threads is greater than 16. For an amount less than this value, it becomes unworthy to add extra running time with the *S3D-ST* choice as long as the *S3D-S* algorithm shares similar or even better performance.

4.4 S3D-Block Mode (S3D-BM) and S3D-Inverted Mode (S3D-IM) results

The average compression rate for the initial 100 frames from both the Upper Bodies and Full Bodies datasets was analyzed for the two variants of the proposed method: S3D-BM and S3D-IM. This average was then compared with the state-of-the-art compression rates, also computed for the first 100 frames of these datasets.

Table 4.5 shows the compression rates achieved with a dyadic decomposition level jump of $k = 2$, where k indicates the number of dyadic decomposition levels skipped.

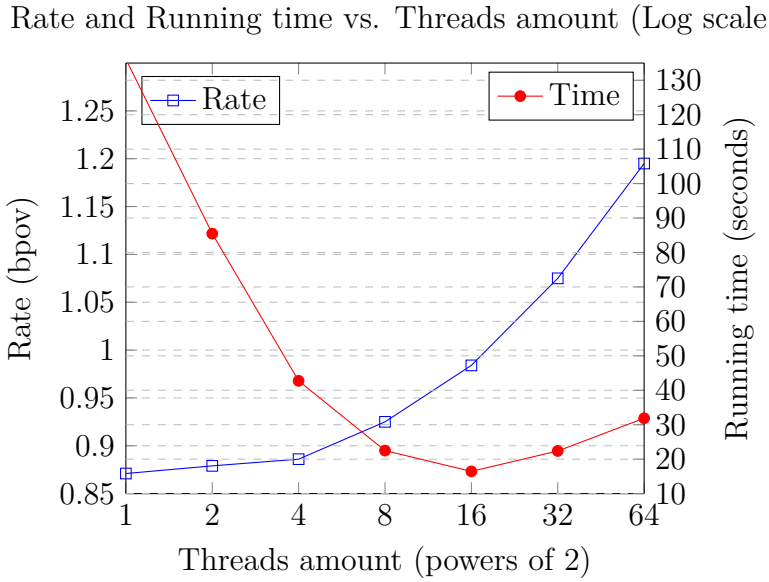


Figura 4.8: Rate and running time plot over different amount of threads for S3D-ST.

Specifically, skipping two levels corresponds to advancing to the decomposition level where the point cloud is divided into four block slices of uniform width.

In the table, the average compression rate across the first 100 frames is compared with state-of-the-art methods. For the Upper Bodies dataset, the results show a clear trend: the proposed variants outperform the TMC13 in average by 5.2%. Notably, CS-S4D [24] introduces a complex approach for selecting the most effective pixel contexts. In comparison, the simpler approach of the S3D-BM achieves performance that is either comparable to or, in some cases, exceeds that of CS-S4D, with an average improvement of 3.16%.

On the other hand, analysis of the Full Bodies dataset reveals that the TMC13 codec still outperforms the other proposals, achieving an average compression rate 4.94% higher than the S3D-BM method. However, the S3D-BM codec still performs better than all other codecs, surpassing the top-performing CS-S4D by 2.41%. These results highlight the efficiency that a simpler variant of the S3D algorithm can offer in terms of compression performance.

Additionally, Table 4.6 presents an interesting comparison between the S3D-BM and S3D-IM variants at $k = 2$. For both the Upper Bodies and Full Bodies datasets, the results show negligible differences in compression rates, despite the Inverted Mode using less contextual data. The performance of both variants is virtually identical when rounded to two decimal places. This finding underscores the adaptability of the algorithm to various scenarios, whether in *real-time acquisition* or other applications, without significant degradation in performance. As such, this flexibility provides users with more options to

select the most suitable codec for their specific needs.

Furthermore, the performance of both the S3D-BM and S3D-IM algorithms was evaluated across a range of k values to assess how the algorithms respond to an increasing number of decomposition blocks, for both the Upper Bodies and Full Bodies datasets. The data, presented in Figures 4.9, 4.10, 4.11, and 4.12, show the average compression rate achieved by these algorithms across the first 100 frames for each dataset, with k varied. A clear inverse relationship between k and compression efficiency emerges from these results: as k increases, compression performance decreases. This trend is expected, as increasing k means decomposing the point cloud into more blocks, which requires transmitting additional bits per subtree in the bitstream, thereby reducing efficiency.

In a *real acquisition* scenario, the amount of k can be interpreted as the availability of storage buffer along the scenario capturing. Considering a device that captures the scene along a single direction in a environment, and its storage is very limited and scarce, it is very important that the codec is capable of compress the little amount of the captured point cloud despite of the lack of great part of the rest. Thus, in the S3D-BM and S3D-IM scenario, this can be easily addressed by increasing the amount of k. We will have more division of blocks of the point cloud, making them narrower, and then adaptable to these scenarios. On an opposite scenario, when the buffer resource availability is abundant, we can have the flexibility of reducing the amount of block by reducing k. And as observed previously in the charts, a more efficient compression performance can be obtained. Therefore, the codec offers more solutions choices to different scenarios, with their pros and cons.

4.5 S3D algorithms benchmark

In the previous results, the aspects and goals of each algorithm were assessed by analyzing their rate, execution time, and the relationship between these metrics and each parameters of the algorithm. This provided a detailed analysis for each algorithm. Now, an overall benchmark is necessary to highlight the strengths and weaknesses of each algorithm.

To simplify the comparison between algorithms, and considering that each may exhibit varying performance based on different parameter values, each algorithm was configured with parameter settings that yielded the best average rate and time across the *upper bodies* and *full bodies*. The process for determining the optimal parameters involved

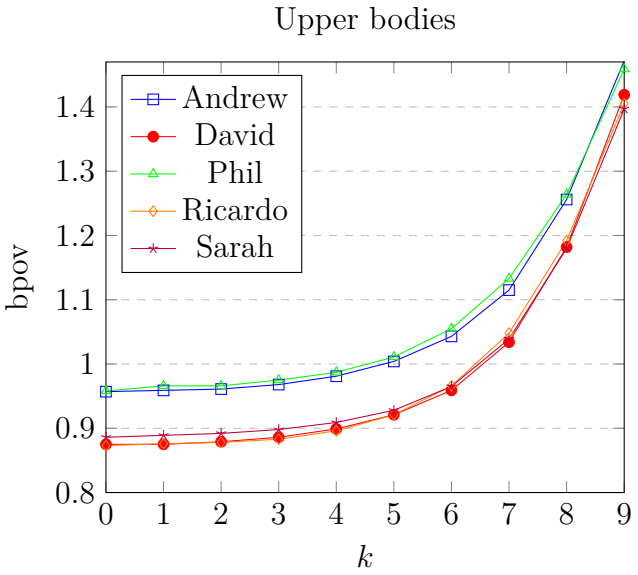


Figura 4.9: Block mode performance for *upper bodies* datasets across various k values.

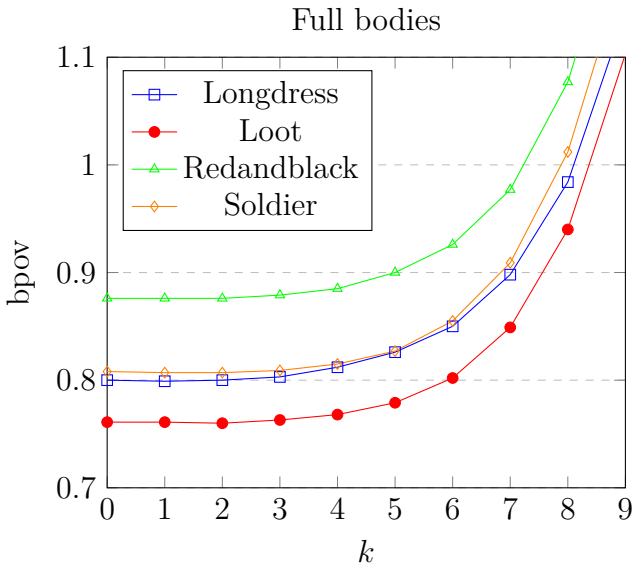


Figura 4.10: Block mode performance for *full bodies* datasets across various k values.

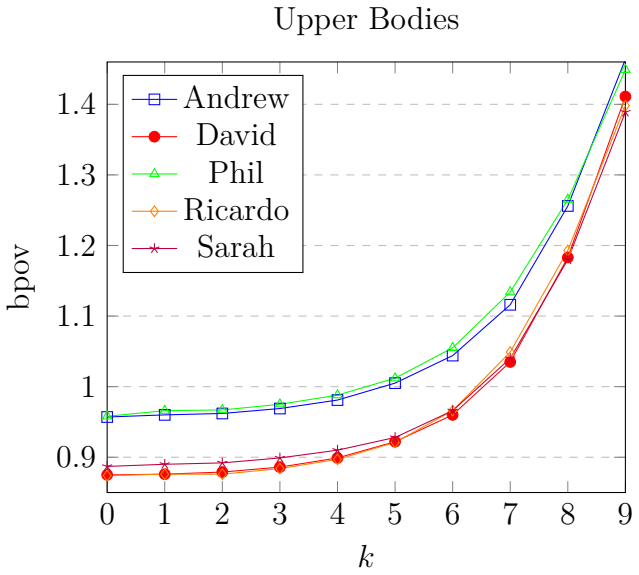


Figura 4.11: Inverted mode performance for each point cloud from *upper bodies* dataset across various k values.

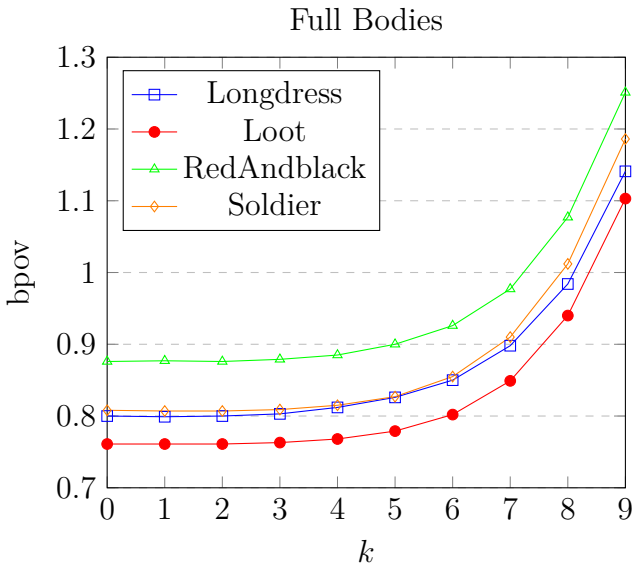


Figura 4.12: Inverted mode performance for each point cloud from *full bodies* dataset across various k values.

Sequence	TMC13 v19 [21]	FRL [22]	S3D Matlab [23]	CS-S4D [24]	S3D-BM
Andrew	1.03	1.00	1.12	**0.95**	0.96
David	0.95	0.94	1.06	0.94	**0.88**
Phil	1.05	1.02	1.14	1.02	**0.97**
Ricardo	0.97	0.93	1.04	0.90	**0.88**
Sarah	0.96	0.95	1.07	0.92	**0.89**
Average	0.99	0.97	1.09	0.95	**0.92**
LongDress	**0.76**	0.86	0.95	0.88	0.80
Loot	**0.71**	0.83	0.92	0.84	0.76
RedAndBlack	**0.84**	0.94	1.02	0.94	0.88
Soldier	0.76	0.88	0.96	**0.65**	0.81
Average	**0.77**	0.88	0.96	0.83	0.81

Tabela 4.5: Comparison of Block mode with k = 2 with state-of-the-art codecs for the first 100 frames from *upper bodies* and *full bodies* datasets, shown in bits per occupied voxel (bpov).

sorting the results by the best rate performance. To reconcile this with time performance, the configuration with the next best rate, if its execution time was at least 15% shorter and the rate difference was less than 0.01, was selected. This criterion served as the stopping condition for the algorithm. This process was repeated in a top-down manner through the results until the condition was met. The final selected configuration was then used for the benchmark.

Although different setups could be selected for benchmarking each codec, this selection method was designed to balance rate performance and execution time, aiming to maximize the benefits of each codec without significant loss in rate. Through this process, the algorithm configurations for the benchmark were determined. Since the S3D C++ codec has no configurable parameters, no selection was necessary. For the S3D-BM and S3D-IM codecs, the optimal setup used $k = 2$. For the S3D-S and S3D-ST codecs, the best configuration was obtained by decomposing the encoding into 4 threads.

The results were based on three repeated measurements taken from the first four frames of both the *upper bodies* and *full bodies* datasets. For each set of measurements, the average rate and time were calculated. The average of the results from the four frames' was then computed for both rate and time. All encodings were performed along the fixed x-axis. The rate and time results are shown in Figures 4.7 and 4.8, respectively. An additional table, which illustrates the relationship between rate and time in a single value (by the product of rate and time), is presented in Figure 4.9.

Several key findings emerge from the results. The S3D C++ codec exhibits the best encoding performance, with the lowest rate for nearly all point clouds, except for Re-

Sequence	S3D-BM	S3D-BMI
Andrew	0.96	0.96
David	0.88	0.88
Phil	0.97	0.97
Ricardo	0.88	0.88
Sarah	0.89	0.89
Average	0.92	0.92
Longdress	0.80	0.80
Loot	0.76	0.76
RedAndBlack	0.88	0.88
Soldier	0.81	0.81
Average	0.81	0.81

Tabela 4.6: Comparison between block mode and its inverted mode for the first 100 frames from from *upper bodies* and *full bodies* datasets, with k equals 2

dAndBlack. However, this performance comes at the cost of execution time, as it has the worst time performance. This trade-off is consistent with the analysis of the S3D-S and S3D-ST results when only one thread is applied. In that case, the encoding approach of the algorithm closely resembles that of S3D C++, with only minor differences. This relationship highlights a key trend: when fewer threads are used, rate performance improves, but execution time worsens. This trend is particularly evident with the S3D C++ codec.

Table 4.9 further supports this analysis showing the *rate.time* product. The higher the product, the worse is the codec performance in terms of rate and time. On the other hand, lower product reflects in a better performance. For S3D C++, which is one of the highest values. This suggests that, although the rate is low, the execution time is significantly longer compared to the other variants, making the product larger.

Furthermore, when comparing the S3D C++ codec with S3D-BM, the same contrast is observed. While S3D C++ offers better rate performance, S3D-BM and S3D-IM achieve execution times that are almost 20 times faster. This difference in execution time can be attributed to the more complex encoding mechanism of S3D C++, which tests the best choice between single mode and dyadic decomposition for each node. In contrast, the simpler and more direct approach of S3D-BM and S3D-IM, using block decomposition followed by single mode, results in significantly faster execution. This disparity is particularly evident in the product *rate.time*, where the S3D-BM and S3D-IM product is nearly 8 times larger than those of S3D-S and S3D-ST, despite the similar rate performance.

Finally, when comparing S3D-BM and S3D-IM with S3D-S and S3D-ST, we observe that they share similar rate performance but differ significantly in execution time. S3D-BM and S3D-IM are approximately 7 times faster in execution. This indicates that,

even with concurrent executions in S3D-S and S3D-ST, the simpler approach of S3D-BM and S3D-IM leads to faster performance. The time efficiency of S3D-BM is particularly noticeable when comparing the product of *rate.time* between the codecs, making the result smaller.

While these results highlight the relative strengths and weaknesses of each codec in terms of rate and time, it is important to note that each algorithm offers more than just an improvement in rate or time performance. They also contribute to the development of point cloud geometry encoding strategies through their unique characteristics. For example, S3D-BM and S3D-IM offer the potential for use in real-time acquisition applications, while S3D-S and S3D-ST explore how geometry decomposition can be combined with multi-threading — an advantage enabled by the S3D approach.

Sequence	S3D C++	S3D-BM	S3D-IM	S3D-S	S3D-ST
Andrew	**0.9799**	0.9978	0.9983	1.0419	1.0489
David	**0.9521**	0.9698	0.9701	1.0208	1.0298
Phil	**1.0086**	1.0244	1.0247	1.0763	1.0856
Ricardo	**0.9035**	0.9153	0.9159	0.9662	0.9741
Sarah	**0.9572**	0.9720	0.9724	1.0235	1.0345
Average	**0.9606**	0.9759	0.9763	1.0227	1.0306
LongDress	**0.8169**	0.8207	0.8209	0.8362	0.8379
Loot	**0.7588**	0.7631	0.7632	0.7747	0.7755
RedAndBlack	0.8866	**0.8821**	0.8823	0.8840	0.8866
Soldier	**0.8055**	0.8083	0.8084	0.8211	0.8227
Average	**0.8140**	0.8185	0.8187	0.8280	0.8307

Tabela 4.7: Rate benchmark of the work's proposed algorithms, average of first 4 frames from each dataset, sampled 3 times, shown in bits per occupied voxel (bpov).

Sequence	S3D C++	S3D-BM	S3D-IM	S3D-S	S3D-ST
Andrew	104.7795	**5.912**	6.1705	36.7165	37.5545
David	127.3900	**7.6375**	7.9095	39.0445	40.0300
Phil	134.6940	**8.2025**	8.5175	40.4785	41.5995
Ricardo	99.7590	**4.4700**	4.6355	31.8935	32.3905
Sarah	129.4885	**7.4315**	7.6920	36.1600	36.9380
Average	126.4220	**6.5705**	6.7830	36.6586	37.3025
LongDress	294.1765	**28.3005**	29.1415	116.2695	120.2100
Loot	267.1380	**22.5470**	23.3000	128.0595	133.3510
RedAndBlack	147.1220	**21.7080**	22.4210	141.2840	147.1220
Soldier	368.9705	**33.8635**	34.9270	167.8515	174.6460
Average	270.6018	**22.7574**	22.9845	138.3664	143.8323

Tabela 4.8: Running time benchmark of the work's proposed algorithms, average of first 4 frames from each dataset, sampled 3 times, shown in seconds (s).

Sequence	S3D C++	S3D-BM	S3D-IM	S3D-S	S3D-ST
Andrew	102.67	5.90	6.16	38.25	39.39
David	121.29	7.41	7.67	39.86	41.22
Phil	135.85	8.40	8.73	43.57	45.16
Ricardo	90.13	4.09	4.25	30.82	31.55
Sarah	123.95	7.22	7.48	37.01	38.21
Average	121.44	6.41	6.62	37.49	38.44
LongDress	240.31	23.23	23.92	97.22	100.72
Loot	202.70	17.21	17.78	99.21	103.41
RedAndBlack	130.44	19.15	19.78	124.90	130.44
Soldier	297.21	27.37	28.23	137.82	143.68
Average	220.27	18.63	18.82	114.57	119.48

Tabela 4.9: Product of rate and time benchmark of the proposed algorithms from the work, average of first 4 frames from each dataset, sampled 3 times, shown in bits per occupied voxel per second ($bpov.s$)

Capítulo 5

Conclusions

This work introduces modifications to the S3D C++ implementation, along with variants based on multi-threading concepts and block decomposition. Initially, the selection of 2D and 3D contexts was reviewed by sampling different results from each combination, using the Microsoft Voxelized Upper Bodies [14] and 8i Voxelized Full Bodies (8iVFB v2) [20]. It was concluded that the optimal setup consisted of eight 2D contexts and nine 3D contexts, which served as the foundation for the subsequent experiments. Additionally, the selection of 3D context pixels was reorganized to improve the readability and clarity of the indexing in the code implementation. With these configurations set for all codecs, each codec was then implemented.

The S3D C++ codec was updated to address pending issues that in the previous version were not covered but were present in the original Matlab implementation, particularly the lack of a single mode and the absence of node testing that compares single mode with dyadic decomposition to determine the more efficient option. The updated S3D C++ outperformed the best state-of-the-art algorithm, CS-S4D, by 5.26% in average compression rate for upper bodies. For full bodies, however, TMC13 still yielded the best results on most datasets, surpassing CS-S4D by an average of 2.41%. Nevertheless, S3D C++ showed superior performance when compared to other variants. Specifically, compared to the original Matlab implementation of S3D, S3D C++ demonstrated a relative improvement of 17.43% for upper bodies and 15.3% for full bodies.

Next, the S3D-S and S3D-ST codecs are proposed as variants based on multi-threading. These codecs aim to maximize the processing capabilities of the device by leveraging recursive and independent subtrees for concurrent processing. For each subtree, a separate thread is assigned for encoding. This enables concurrent processing of each tree by its respective thread, optimizing the overall encoding process by distributing the CPU workload across multiple processes, rather than treating the entire tree as a single process. This approach aims to reduce encoding time, though it sacrifices performance by losing

context between the subtrees.

Two variants are proposed: S3D-S and S3D-ST. In S3D-S, the encoding is performed in the same manner as the original S3D for each subtree, with the results aggregated into a bitstream. In S3D-ST, each subtree is encoded similarly to S3D, but the root node is ignored in the encoding process, using only its context. To achieve this, a reduced S3D encoding is performed at the top of the dyadic decomposition tree of the point cloud, where the leaf nodes represent the root nodes of the subtrees.

An analysis of the relationship between time performance improvements and trade-offs in rate performance with increasing threads is presented. In terms of time, significant reductions in execution time are observed with the increase in threads for both S3D-S and S3D-ST, with a 60% reduction when using 4 threads compared to the original S3D. However, the rate performance deteriorates as the number of threads increases. Notably, for thread counts below 16, the performance of both algorithms remains similar, but beyond this threshold, S3D-ST demonstrates higher efficiency. This was expected, as S3D-ST encodes the root nodes of subtrees through a reduced S3D process at the top of the complete tree, rather than encoding large-scale silhouettes in their entirety.

The final group of proposed codecs is based on block decomposition: S3D-Block Mode (S3D-BM) and S3D-IM. These codecs introduce a new geometric decomposition model for point clouds, in addition to those already present in S3D (dyadic decomposition and single mode). The block decomposition model directly partitions the point cloud into blocks of fixed size, where the size is a power of 2. Compared to dyadic decomposition, this approach effectively skips directly to a specific level in the decomposition tree. A fixed parameter k is introduced to the algorithm, representing the number of levels skipped from the root node in the decomposition tree, resulting in 2^k blocks. While S3D-BM employs a top-down approach, S3D-IM adopts a bottom-up approach, making it suitable for real-time acquisition contexts.

When comparing S3D-BM ($k = 2$) with state-of-the-art codecs on the upper bodies dataset, it demonstrates an average outperformance of 5.2% relative to TMC13. Additionally, compared to CS-S4D, it achieves an average improvement of 3.16%. On the full bodies dataset, however, TMC13 of MPEG remains the top performer, outperforming S3D-BM by an average of 4.94%. Nonetheless, S3D-BM surpasses other state-of-the-art codecs, particularly CS-S4D, which is the best-performing among them, by an average of 2.41%. This is a notable result, given that CS-S4D employs a more complex approach compared to the simplicity of S3D-BM, highlighting the effectiveness of the proposed method.

The comparison between S3D-BM and S3D-IM reveals that both variants exhibit similar rate performance. This similarity indicates the versatility of these codecs for diffe-

rent scenarios, such as real-time acquisition or simpler contexts. Finally, the relationship between rate performance and the parameter k in S3D-BM and S3D-IM is plotted. The results show that as k increases (i.e., as the number of blocks grows), rate performance worsens. This is due to the increased number of blocks to be encoded and sent in the bitstream. The choice of k can be linked to device buffer constraints: with higher storage capacity, fewer blocks (2^k) are needed, improving rate performance. Conversely, in scenarios with limited storage capacity, higher k values (smaller blocks) are more appropriate, albeit at the cost of reduced rate efficiency.

After analyzing the specific characteristics of each S3D variant, a benchmark was conducted to assess the rate and time performance of all the algorithms. For this purpose, each result from the codec were based on the best combination of parameters, balancing the optimal rate with minimal running time. With the best setups identified for each S3D codec, the results were collected. In addition to comparing rate and time individually, a new metric was introduced, which is the product between rate and the running time, in order to better understand the balance between both metrics. This helps to avoid favoring codecs that either have a low rate with long execution times or short execution times with high rates.

The analysis showed that S3D-BM does not achieve the best rate, but it strikes a favorable balance with a good rate and fast running time. While these metrics help identify the best and worst performers, the main contribution of each codec lies in its unique applicability and characteristics. S3D-C++ delivers a superior rate compared to state-of-the-art algorithms. S3D-BM and S3D-IM offer a simpler approach to point cloud decomposition, based on the S3D method, and their decomposition mechanism makes them well-suited for real-time acquisition contexts. Finally, the thread-based S3D-S and S3D-ST codecs introduce a novel method for geometry point cloud encoding while exploring the use of multi-threading in their decomposition approach. The range of applicability and the concepts explored by these variants are the key contributions of this work.

Referências

[1] Cui, Li, Rufael Mekuria, Marius Preda e Euee S. Jang: *Point-cloud compression: Moving picture experts group's new standard in 2020*. IEEE Consumer Electronics Magazine, 8(4):17–21, 2019. 1, 2, 12, 13

[2] Mekuria, Rufael, Kees Blom e Pablo Cesar: *Design, implementation, and evaluation of a point cloud codec for tele-immersive video*. IEEE Transactions on Circuits and Systems for Video Technology, 27(4):828–842, 2017. 1

[3] Queiroz, Ricardo L. de e Philip A. Chou: *Compression of 3d point clouds using a region-adaptive hierarchical transform*. IEEE Transactions on Image Processing, 25(8):3947–3956, 2016. 1, 2, 16, 24

[4] Hackel, Timo, N. Savinov, L. Ladicky, Jan D. Wegner, K. Schindler e M. Pollefeys: *SEMANTIC3D.NET: A new large-scale point cloud classification benchmark*. Em *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume IV-1-W1, páginas 91–98, 2017. 2

[5] Huang, Yan, Jingliang Peng, C. C. Jay Kuo e M. Gopi: *A generic scheme for progressive point cloud coding*. IEEE Transactions on Visualization and Computer Graphics, 14(2):440–453, 2008. 2, 24

[6] MPEG: *Call for proposals for point cloud compression v2*. ISO/IEC JTC1/SC29/WG11 MPEG2017/N16763, April 2017, Hobart, AU. 2

[7] Rosário, Rodrigo e Eduardo Peixoto: *Intra-frame compression of point cloud geometry using boolean decomposition*. Em *2019 IEEE Visual Communications and Image Processing (VCIP)*, páginas 1–4, 2019. 5

[8] Peixoto, Eduardo: *Intra-frame compression of point cloud geometry using dyadic decomposition*. IEEE Signal Processing Letters, 27:246–250, 2020. 5, 7, 11, 25, 31

[9] Freitas, Davi R., Eduardo Peixoto, Ricardo L. de Queiroz e Edil Medeiros: *Lossy point cloud geometry compression via dyadic decomposition*. Em *2020 IEEE International Conference on Image Processing (ICIP)*, páginas 2731–2735, 2020. 5, 26

[10] Peixoto, Eduardo, Edil Medeiros e Evaristo Ramalho: *Silhouette 4d: An inter-frame lossless geometry coder of dynamic voxelized point clouds*. Em *2020 IEEE International Conference on Image Processing (ICIP)*, páginas 2691–2695, 2020. 5

[11] Alves, Lucas Martins: *Paralelização do algoritmo de compressão de nuvem de pontos silhouette 3d*. Relatório Técnico, Trabalho de Conclusão de Curso. (Graduação em Engenharia Elétrica) - Universidade de Brasília. Orientador: Eduardo Peixoto Fernandes da Silva, 2022. 7, 8, 11

[12] Komatsu, Otho T., Edil Medeiros, Lucas M. Alves e Eduardo Peixoto: *Multithreaded algorithms for lossless intra compression of point cloud geometry based on the silhouette 3d coder*. Em *2023 IEEE International Conference on Image Processing (ICIP)*, páginas 1880–1884, 2023. 8, 40

[13] Loop, Charles, Cha Zhang e Zhengyou Zhang: *Real-time high-resolution sparse voxelization with application to image-based modeling*. Em *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, página 73–79, New York, NY, USA, 2013. Association for Computing Machinery, ISBN 9781450321358. `https://doi.org/10.1145/2492045.2492053`. 12

[14] *Jpeg pleno database: Microsoft voxelized upper bodies - a voxelized point cloud dataset*. `http://plenodb.jpeg.org/pc/microsoft`. Accessed: 2021-08-29. 13, 52, 71

[15] *Ply - polygon file format*. `http://paulbourke.net/dataformats/ply/`. Accessed: 2021-08-29. 14

[16] Schnabel, Ruwen e Reinhard Klein: *Octree-based point-cloud compression*. Em *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, página 111–121, Goslar, DEU, 2006. Eurographics Association, ISBN 3905673320. 16, 24

[17] Krivokuća, Maja, Philip A. Chou e Maxim Koroteev: *A volumetric approach to point cloud compression–part ii: Geometry compression*. IEEE Transactions on Image Processing, 29:2217–2229, 2020. 24

[18] Tanenbaum, Andrew S.: *Operating Systems: Design and Implementation*. Prentice Hall, 2006. 35

[19] Peixoto, E., E. Medeiros, E. Lemos, E. Albuquerque, O.T. Komatsu e R. Borba: *S3d c++ implementation codec*. `https://github.com/pointcloud-unb/SilhouetteCoder`. Accessed: 2024-01-05. 48

[20] d'Eon, E., B. Harrison, T. Myers e P. A. Chou: *8i Voxelized Full Bodies, version 2 – A Voxelized Point Cloud Dataset*. Relatório Técnico, ISO/IEC JTC1/SC29/WG11 m40059 ISO/IEC JTC1/SC29/WG1 M74006 Geneva, Switzerland, 2017. 52, 61, 71

[21] 3DG: *G-PCC codec description v4*. Relatório Técnico, ISO/IEC JTC 1/SC 29/WG 11 input document w18673, 2019. 57, 67

[22] Tzamarias, Dion E. O., Kevin Chow, Ian Blanes e Joan Serra-Sagristà: *Fast run-length compression of point cloud geometry*. IEEE Transactions on Image Processing, 31:4490–4501, 2022. 57, 67

[23] Peixoto, Eduardo: *Intra-frame compression of point cloud geometry using dyadic decomposition*. IEEE Signal Processing Letters, 27:246–250, 2020. 57, 67

[24] Ramalho, Evaristo, Eduardo Peixoto e Edil Medeiros: *Silhouette 4d with context selection: Lossless geometry compression of dynamic point clouds*. IEEE Signal Processing Letters, 28:1660–1664, 2021. 57, 63, 67

[25] Loop, C., Q. Cai, S. O. Escolano e P. A. Chou: *Microsoft Voxelized Upper Bodies – A Voxelized Point Cloud Dataset*. Relatório Técnico, ISO/IEC JTC1/SC29/WG11 m38673 ISO/IEC JTC1/SC29/WG1 M72012, Geneva, Switzerland, 2016. 61