DISSERTAÇÃO DE GRADUAÇÃO

Compressão de Nuvens de Pontos: Transposição do Algoritmo Silhouette 3D da Linguagem MATLAB para a Linguagem C++

Estevam Galvão Albuquerque

Graduação em Engenharia de Computação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA/CIÊNCIA DA COMPUTAÇÃO
FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE GRADUAÇÃO

Compressão de Nuvens de Pontos: Transposição do Algoritmo Silhouette 3D da Linguagem MATLAB para a Linguagem C++

Estevam Galvão Albuquerque

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

Banca Examinadora

Prof. Eduardo Peixoto Fernandes da Silva, Ph.D, FT/UnB
Orientador _____

Prof. José Edil Guimarães de Medeiros, Ph.D, ENE/UnB
Examinador Interno _____

Prof. Gustavo Luiz Sandri, Ph.D, IFB
Examinador externo _____

FICHA CATALOGRÁFICA

ALBUQUERQUE, ESTEVAM GALVÃO
Compressão de Nuvens de Pontos:Transposição do Algoritmo Silhouette 3D da Linguagem MATLAB para a Linguagem C++ [Distrito Federal] 2021.
xvi, 42 p., 210 x 297 mm (ENE/FT/UnB, Graduação, Engenharia de Computação, 2021).
Dissertação de Graduação - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Nuvem de pontos
2. Codificação aritmética
3. Compressão de geometria
4. Codificação intra frame

I. ENE/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ALBUQUERQUE, E. G. (2021). *Compressão de Nuvens de Pontos:Transposição do Algoritmo Silhouette 3D da Linguagem MATLAB para a Linguagem C++*. Dissertação de Graduação, Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, 42 p.

CESSÃO DE DIREITOS

AUTOR: Estevam Galvão Albuquerque
TÍTULO: Compressão de Nuvens de Pontos:Transposição do Algoritmo Silhouette 3D da Linguagem MATLAB para a Linguagem C++.
GRAU: Graduação em Engenharia de Computação ANO: 2021

Estevam Galvão Albuquerque
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

Dedico este trabalho aos meus pais, Rildenia Maria e Estevam Manuel, por todo o amor e carinho que me deram e ao meu orientador, Eduardo Peixoto, por ter me proporcionado a oportunidade de participar deste trabalho.

AGRADECIMENTOS

Agradecer faz parte do processo de respeitar o presente e compreender o passado. Por mais que eu não tenha o costume de agradecer todas as coisas boas que pude presenciar, as pessoas incríveis que pude conhecer e as incontáveis risadas que pude dar até este exato momento da minha vida, hoje eu gostaria de fazê-lo. Gostaria de agradecer aos meus pais, Rildenia Maria e Estevam Manuel, que se sacrificaram em prol de fornecer a mim a melhor qualidade de vida que eles podiam. A minha mãe, especificamente, por ter me ensinado a ser uma pessoa carinhosa e me educado com liberdade. As todas as pessoas que eu tive o prazer de chamar de amigo até hoje e a Universidade de Brasília por ter me proporcionado uma das melhores épocas da minha vida, de mais esforço e amadurecimento.

Obrigado.

RESUMO

Point Clouds têm se revelado como uma das mais promissoras tecnologias para representação de conteúdo 3D dos últimos anos. Descritas por sua geometria e características adicionais, como cor ou refletância, quando elevadas a um grande nível de detalhamento podem se tornar estruturas de manuseio e armazenamento custoso. O meio acadêmico têm alavancado esforços em desenvolver métodos de compressão para essas estruturas visto a quantidade crescente de pesquisas a respeito desse assunto. Em 2020, o professor da Universidade de Brasília, Eduardo Peixoto, propôs uma técnica de compressão de geometria para Point Clouds inspirado em algoritmos clássicos de compressão de imagens binárias. Implementado em um ambiente experimental não ideal para esse nível de aplicação, o algoritmo ainda obteve resultado significativos, desempenhando melhor que modelos do estado-da-arte apresentados naquele ano. Dessa forma, com resultados estimulantes porém prejudicados por limitações técnicas em detrimento do ambiente optado, a migração do algoritmo proposto do ambiente MATLAB para a linguaguem de programação C++, atrelado a reestruturação do código, pretende incrementar o nível do projeto, a ponto de torná-lo competitivo.

ABSTRACT

Point Clouds have proven to be one of the most promising technologies for 3D content representation in recent years. Described by their geometry and additional features such as color or reflectance, when raised to a high level of detail they can become structures of costly handling and storage. The academia has put effort to develop compression methods for these structures since the growing amount of research on this subject. In 2020, the professor at the University of Brasilia, Eduardo Peixoto, proposed a geometry compression technique for Point Clouds inspired by classical binary image compression algorithms. Implemented in an experimental environment not ideal for this kind of application, the algorithm still obtained significant results, performing better than the state-of-the-art models presented that year. So that, with stimulating results, however, hampered by technical limitations to the detriment of the chosen environment, the migration of the proposed algorithm from the MATLAB environment to the C++ programming language, coupled with code restructuring, intends to increase the level of the project, to the point of making it competitive.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

O capítulo a seguir abordará as principais motivações, objetivos, expectativas e o contexto o qual serviu de inspiração para o projeto apresentado.

1.1 CONTEXTO

É recorrente, tanto no âmbito acadêmico como no comercial, confeccionar um protótipo a partir de uma ideia promissora através de um ambiente de rápido desenvolvimento, como, por exemplo, uma linguagem de programação de alto nível. Sob esse viés, conforme o protótipo prospera e barreiras técnicas provenientes do ambiente escolhido são encontradas, é comum optar pela mudança para um que seja mais condizente com os objetivos do projeto. Tratando-se da migração de um algoritmo, essa não corresponde a uma tarefa trivial, visto que envolve o reestudo de ponta-a-ponta do algoritmo em questão, levando em consideração seu cenário de atuação. De todo modo, a migração, se articulada de maneira eficiente, trás benefícios valiosos ao futuro código vigente como manutenção, novas funcionalidades ou desempenho.

Dito isso, a demanda sobre objetos tri-dimensionais (3D) tem expandido notavelmente em detrimento dos diversos campos de estudos interessados em interpretar e reproduzir espaços através de imagens digitais 3D, desde representação de seres humanos até construções e desenvolvimento de carros autônomos. Entretanto, essas espécies de aplicações exigem um grande nível de detalhamento, onde as malhas poligonais convencionais são geralmente muito complicadas para representarem esses dados. Nesse sentido, as Point Clouds revelam ser uma poderosa alternativa pois são amostradas diretamente na superfície do objeto 3D e não requerem informações sobre a conectividade entre vértices. A simplicidade da distribuição de pontos nos retorna uma renderização superior sobre modelos 3D consideravelmente grandes [7].

Atrelado à esse viés, ao trabalhar com dados no geral, podemos afirmar que é pretendido transmití-los ou armazená-los de alguma forma, objetivo o qual pode revelar-se bastante desafiador conforme esses dados aumentam de tamanho, principalmente ao falarmos sobre Point Clouds, estas podem crescer até bilhões de pontos dependendo do nível de detalhamento. Isto posto, a compressão de dados torna-se uma prática imprescindível nesse cenário, pois mesmo que as tecnologias de armazenamento e transmissão tenham melhorado bastante, estes ainda são recursos limitados e custosos.

Por essa razão, novas tecnologias têm sido desenvolvidas, buscando fomentar a capacidade de armazenamento, consolidando a área de compressão de sinais. Dessa forma, surge um dos principais grupos vinculados a essa área, o MPEG (Moving Picture Experts Group), um grupo de trabalho responsável pelo desenvolvimento de padrões internacionais para compressão, descompressão, processamento e representação codificada de imagens em movimento, áudio e suas combinações. Em um mundo onde a tecnologia da informação, produtos eletrônicos, produtos de entretenimento e telecomunicações convergem de várias maneiras, incorporando tecnologias cada vez mais sofisticadas, a exigência de padrões disponíveis em tempo hábil revela-se indispensável. Dessa forma, o MPEG fornece um mecanismo comprovado para

trazer resultados de pesquisa em padrões que promovam inovação para o benefício de todos [8].

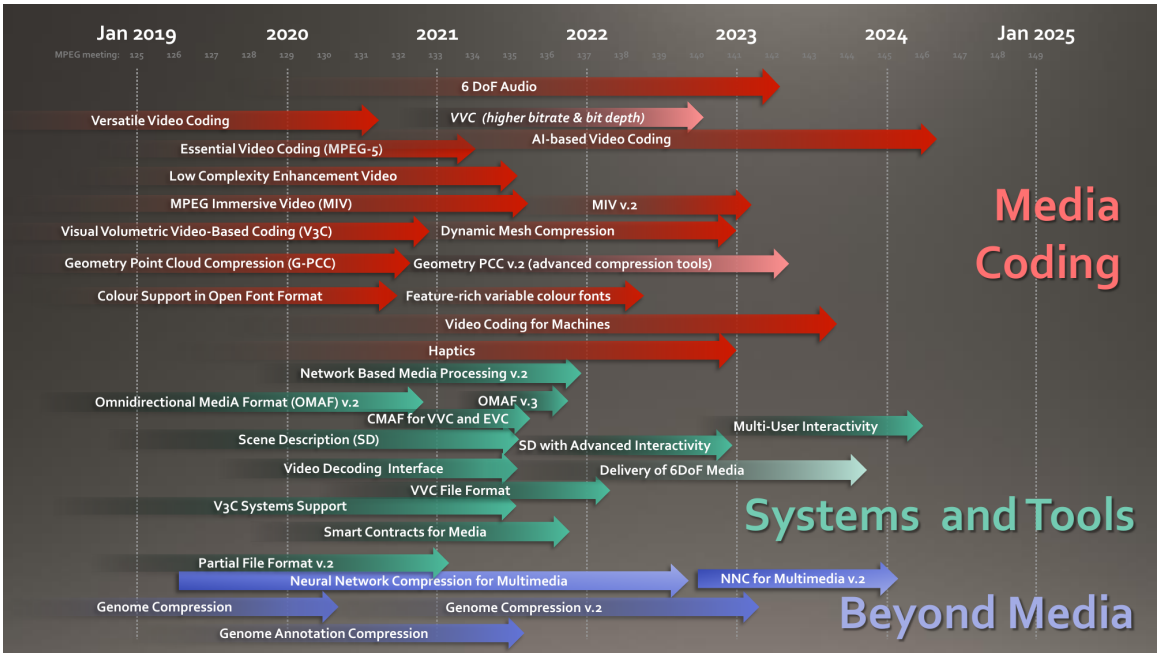

Figura 1.1: Roteiro de desenvolvimento de novos padrões do MPEG.

1.2 MOTIVAÇÃO

Por sua versatilidade e eficiência em aplicações, pesquisas e propostas de algoritmos de compressão dessas Point Clouds têm avançado consideravelmente [1]. Dentre essas propostas, no ano de 2020 foi proposto um algoritmo que consistia em uma compressão geométrica de Point Clouds inspirada em algoritmos clássicos de compressão de imagens binárias [9], o qual se sobressaiu perante os algoritmos estado-da-arte apresentados nessa conferência. O algoritmo elaborado foi desenvolvido em MATLAB, ferramenta a qual se caracteriza por ser bastante experimental e menos adequada para uma proposta buscando eficiência a nível de aplicação. Tanto o tempo de execução, quanto análises mais profudas sobre o algoritmo foram prejudicadas por limitações da ferramenta. Desse modo, enxergou-se uma oportunidade de elevar o desempenho do algoritmo proposto a outro nível, ao desenvolvê-lo em linguagem que não possua tais limitações.

Visto isso, conciliando a continuidade nos estudos sobre o algoritmo apresentado, a busca por melhorias deste, o estudo dos outros métodos também considerados estado da arte e sobre Point Clouds em si, o projeto proposto se baseia na modelagem e reimplementação deste algoritmo na linguagem de programação C++, visando principalmente a evolução sobre o tempo de execução e desempenho. Assim, a motivação deste trabalho surge da necessidade de aprimorar um algoritmo promissor da área de compressão de Point Clouds, atrelado a uma análise das vantagens de C++ perante ao MATLAB para o estudo apresentado. O propósito principal deste projeto é, então, analisar o funcionamento de um algoritmo em MATLAB, entender seus requisitos e implementar uma proposta superior em desempenho na linguagem C++.

1.3 PROBLEMA

O MATLAB, por se tratar de uma ferramenta de linguagem de alto nível interpretada, torna mais produtivo o caminho do desenvolvedor em diversos aspectos durante o desenvolvimento, agilizando o epílogo do projeto desejado em prol de tomar decisões por este mesmo desenvolvedor. Para que o programa seja executado por uma aplicação, este necessita estar em uma linguagem que o processador compreenda, o que se resume a linguagem de máquina, baseada no sistema binário (0 e 1). Logo, todo código em qualquer linguagem que não de máquina, será reduzida a linguagem de máquina antes de ser operada pelo processador. A linguagem interpretada, por sua vez, para ser compreendida no fim do processo, demanda que outro software atue como tradutor durante sua realização, fazendo com que o tempo de execução cresça consideravelmente. Dessa forma, é comum encontrar défices no que diz respeito ao manuseio de memória e tempo de execução em códigos confeccionados em MATLAB. Assim, conforme as Point Clouds codificadas no projeto cresciam em tamanho, esses défices se revelavam cada mais custosos a ponto de tornar o tempo de execução impraticável em relação ao que o algoritmo foi proposto.

À vista disso, em função de obter resultados que possam competir com os apresentados ao MPEG, a implementação do algoritmo aplicado a uma linguagem compilada e que dependa mais do desenvolvedor no que diz respeito ao manuseio de memória, configura uma possível saída para sanar a necessidade de reduzir custos de memória e tempo de execução do projeto. Dito isso, a migração do software para a linguagem de programação C++ buscar explorar as características desejadas presentes na linguagem, aliado à estruturas do paradigma orientado a objeto a fim de concretizar a solução idealizada.

1.4 EXPECTATIVAS

As expectativas para esse estudo são a respeito do exercício e desenvolvimento sobretudo de habilidades como trabalho em equipe, capacidade de analizar e compreender um projeto complexo em andamento e compartilhar protocolos de projeto com uma equipe de desenvolvedores, como estipular um controle de versionamento e seguir um padrão de projeto estabelecido. Além disso, realizar com sucesso a migração de um software para outra linguaguem com o objetivo de superar o desempenho da implementação anterior e contribuir na elaboração de uma proposta competitiva de padrão de codificação para o MPEG. Caso não seja possível atingir esses objetivos, espera-se ao menos obter uma versão funcional de forma que possa servir de base para a continuação do projeto e novas otimizações.

2 REVISÃO BIBLIOGRÁFICA

O objetivo desse capítulo é rever conceitos já disseminados na literatura e elucidar o contexto do trabalho apresentado a fim de compreende-lo melhor.

2.1 POINT CLOUD

Uma Point Cloud pode ser definida como um conjunto de pontos de três dimensões em um espaço Euclidiano, onde cada ponto é descrito por suas coordenadas (x, y, z) e pode carregar atributos adicionais como cor, refletância, textura, entre outros dependendo da necessidade de cada aplicação. Ao reunir um grande número de tais pontos espaciais individuais em um único conjunto de sistema comum, pode-se capturar a geometria de objetos 3D inteiros, consequentemente, uma Point Cloud. As Point Clouds podem ser classificadas com estáticas, representadas pela captura de apenas um instânte no tempo ou um quadro, e dinâmicas, constituída por uma sequência de Point Clouds estáticas, cada uma em seu próprio quadro. Naturalmente, no caso das dinâmicas, a quantidade de pontos em cada quadro pode variar.

Figura 2.1: Vista renderizada de Point Cloud Ricardo9 frame 37

Dito isso, em decorrência da sua possibilidade de prover as coordenadas espaciais das superfícies observadas e a possibilidade de visualizar um ambiente 3D em um display 2D, dando ao usuário a possibilidade de trocar o ponto de vista ou navegar na cena, Point Clouds têm sido reconhecidas como a aplicação mais apropriada para visualização de dados 3D, desde na modelagem 3D de uma pessoa até no mapeamento de ambientes urbanos de grande escala [10][11]. Os pontos 3D medidos consistem nas coordenadas espaci-

ais diretas de superfícies geométricas, permitindo assim, uma simplificação significativa dos processos de modelagem de superfície e reconstrução geométrica [12].

Figura 2.2: Pontos de vista de uma Point Cloud. Figura cortesia de [1]

Em contraste com os já conhecidos triângulos de malhas, as Point Clouds não possuem a necessidade de armazenar ou manter informação sobre as conexões internas do objeto 3D. Sua simplicidade, sua flexibilidade e sua robusta capacidade de representação integram o crescente conjunto de razões atreladas a popularidade dessas estruturas nos mais diversos contextos. Também é válido citar quanto espaço as Point Clouds conquistaram no que diz respeito ao ambiente acadêmico, como temática de pesquisas científicas, devido ao surgimento de sensores de baixo-custo como o Kinect [13] e câmeras ToF (Time of flight) [14]. Embora tais sensores tenham se tornado muito populares, presentes até mesmo em celulares, como o Apple's iPhone X e o Sony's Xperia XZ1 (para Point Clouds mais simples, com centenas de pontos), muitas vezes a obtenção dos pontos que constituem a Point Cloud não é simples devido ao ruído proveniente das limitações desses mesmos sensores. Atualmente existem várias técnicas de filtragem com o objetivo de tratar e suavizar esses ruídos. [15].

Dessa forma, após a coleta dos dados que constituem a Point Cloud, estes ainda necessitam ser tratados e por mais que existam aplicações que utilizam Point Clouds brutas, estas, normalmente são desestruturadas, irregulares, possuem redundância para múltiplas fontes e a recuperação de uma grande quantidade de dados não é pouco trivial [3]. Dessa forma, organizar e estruturar Point Clouds se revela um método necessário e efeciente para realizar qualquer manuseio sobre dados destas. Encontram-se, na literatura, algumas abordagens para esse tipo de estratégia, como imagens de profundidade, malhas 3D, Point Clouds brutas e rede de voxels[3].

2.1.1 Voxalização

Como as amostras de Point Clouds utilizadas no presente estudo são voxalizadas, esta abordagem será mais aprofundada. Assim como em imagens, onde unidades de área são chamadas de *pixel(s) (picture element)*, em Point Clouds, podemos ter unidades de volumes chamadas de *voxel(s) (volume element)*, os quais traduzem um modelo de representação 3D topologicamente explícita e rica em informação construída através de uma grade estruturada. Sob um ponto de vista conceitual, voxels possuem a geometria de um cubo, com seis faces, oito vertices e doze arestas, no entanto, esses geralmente não são armazenados como

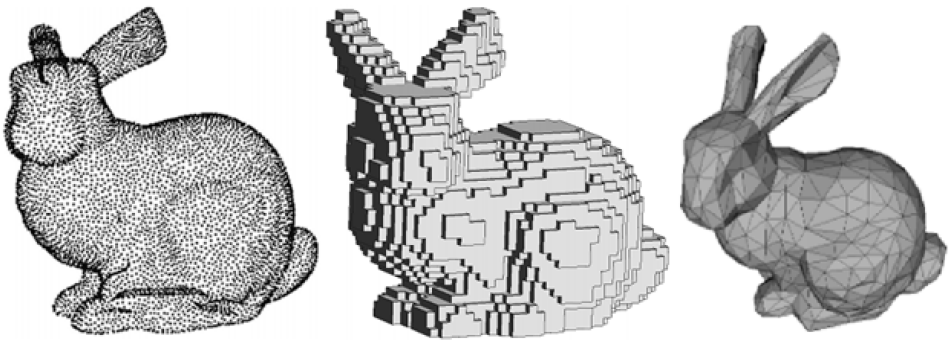

Figura 2.3: Principais representações 3D do coelho de Stanford. **Esquerda:** Representação em Point Cloud. **Meio:** Representação em Voxels. **Direita:** Representação em malhas. Figura cortesia de [2]

estruturas poliédricas de fato. Em vez disso, a representação baseada em voxels utiliza apenas o ponto central ou as vértices para representar o cubo por completo. Sob essa lógica, a voxalização de uma Point Cloud consiste em revestir esta com uma grade de voxels estruturada e, assim, estimar as geometrias e atributos dos pontos dentro de cada voxel como ilustrado na Figura 2.4. Assim, toda a Point Cloud é divida por voxels, onde os pontos dentro de um determinado voxel são representados por esse mesmo voxel. Comparado a outros métodos de estruturação de dados, a voxelização resulta em uma representação mais simples de objetos complexos, eliminando ainda mais custos operacionais de computação [3].

Figura 2.4: Representação baseada em voxel de uma nuvem de pontos: (a) Pontos originais; e (b) Voxels renderizados. Figura cortesia de [3]

Na Fig 2.5, pode-se ter uma visão geral de algumas das aplicações comuns relacionadas a representação de Point Clouds utilizando voxels e faz-se possível perceber as múltiplas funções que os voxels podem desempenhar ao representarem estruturas 3D. Dessa forma, em determinadas aplicações, Point Clouds podem crescer até serem compostas por milhares de pontos dependendo da quantidade de informação desejada para representar sua estrutura. Conforme estas aumentam em tamanho e detalhes, mais custoso se torna o seu manuseio e, portanto, mais estreita se torna a relação entre a necessidade de comprimí-las e o desenvolver das teclonogias atreladas a essas estruturas.

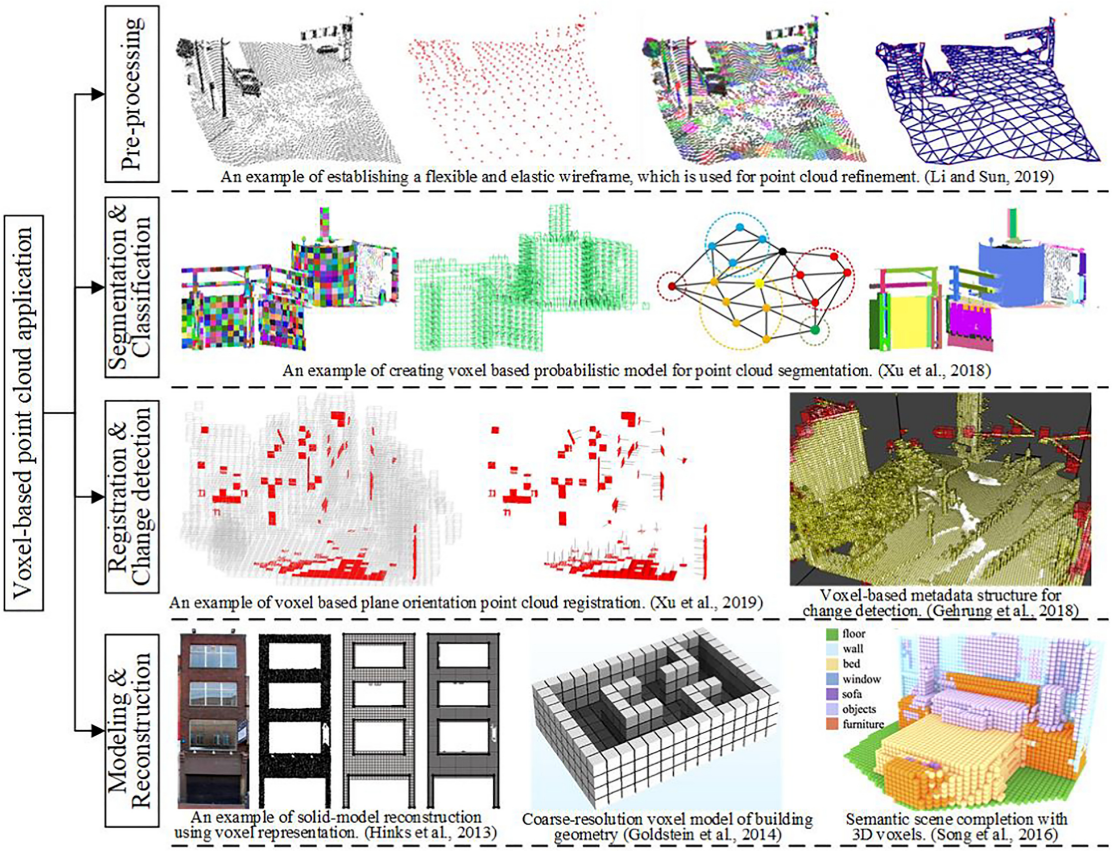

Figura 2.5: Exemplos de aplicações que utilizam representações baseadas em voxels. Figura cortesia de [3]

2.2 COMPRESSÃO DE SINAIS

O **conceito** de compressão pode ser entendido como codificar um sinal utilizando menos informação que sua representação original, onde essa compressão pode ser classificada como sem perdas ou com perdas. No cenário da compressão sem perdas, a representação final reconstruída é exatamente igual à original, enquanto no cenário da compressão com perdas, temos uma representação reconstruída equivalente porém não idêntica à original [16]. O sinal alvo da compressão pode retratar desde simples arquivos binários até arquivos de vídeo completos, no qual o tipo de compressão recomendada para cada sinal está intimamente relacionada ao uso desse sinal. Isso se deve ao fato de que a compressão pode aproveitar do modo com que o usuário final enxerga o sinal em si e suas deformações. A compressão com perdas, por exemplo, é comumente encontrada em fotos digitais, arquivos de áudio e vídeos justamente pelo fato de que o cérebro humano tolera pequenas deformações nessas mídias sem comprometer o seu significado [16]. Por outro lado, a compressão sem perdas é usualmente utilizada em aplicações que não podem tolerar erros, como textos, códigos, imagens médicas, entre outros. Em contrapartida, a compressão com perdas usualmente entrega alguma distorção porém garante uma taxa de compressão muito superior quando comparada à sem perdas, a qual comprime menos porém mantém a qualidade intacta.

A importância do presente tópico é ilustrada a todo momento no dia-a-dia. Atualmente, é possível fazer ligações, escutar músicas, assistir filmes, navegar pela internet graças a compressão de dados. Até

mesmo a formação do alfabeto braille é influenciado pelo estudo da compressão de dados. Dessa forma, é presenciado, nos dias de hoje, uma transformação sobre como os meios de comunicação têm sido utilizados, principalmente quando se trata da quantidade massiva de mídias compartilhadas. A compressão de sinais é um dos estudos habilitadores dessa dita revolução das mídias [16]. Sendo uma ciência tão importante, revela-se imprescindível haver uma maneira de estipular a eficiência dos métodos/algoritmos de compressão propostos.

Nesse sentido, duas características fundamentais são elencadas ao falar de **desempenho**, são elas: a taxa de compressão e a distorção. Entretanto, tratando-se de um algoritmo, além dessas duas últimas, ainda podemos avaliar o desempenho deste através de aspectos convencionais como quanta memória é utilizada durante sua execução ou o quão rápido o algoritmo desempenha. À vista disso, podemos definir a taxa de compressão como a relação entre o tamanho dos dados originais, antes de serem comprimidos, e o tamanho dos dados após a compressão. Também pode-se representar a taxa de compressão expressando a redução na quantidade de dados necessária como uma porcentagem do tamanho dos dados originais.

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

ou

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

Onde CR representa a taxa de compressão, S_o representa o tamanho original dos dados em bits e S_c representa o tamanho dos dados comprimidos. No contexto de compressão com perdas, o produto final difere dos dados originais, onde essa diferença é normalmente denominada distorção. Na época presente, a expressão mais utilizada ao buscar mensurar essa distorção matematicamente é a medida de erro quadrático. Assim, se Xn é a fonte de saída e Yn é a sequência reconstruída, então a medida de erro quadrático é dada por

$$d(Xn, Yn) = 1/n \sum_{n} (Xn - Yn)^2. \tag{2.3}$$

Entretanto, analizar a distorção resultante do processo de compressão, revela-se uma tarefa mais complexa pois a análise está intimamente relacionada ao usuário final. A melhor abordagem para avaliar a distorção em uma imagem comprimida de uma casa para uma propaganda de imóveis, talvez seja indagar um agente imobiliário sobre o que ele pensa do resultado. Por outro lado, se a imagem comprimida vier de um satélite a fim de ser processada por uma máquina com o propósito de obter informações sobre objetos na imagem, talvez a melhor abordagem seja verificar como a distorção afeta o funcionamento vigente da máquina. De maneira geral, é possível afirmar que a abordagem ideal para analizar uma distorção, seria sempre fazer uso do usuário final para inspecionar a qualidade e forcenecer o parecer necessário sobre o resultado analizado [16]. Posto isso, o presente estudo abordará a implementação de um algoritmo de compressão sem perdas para Point Clouds. Ainda assim, para compreender melhor a compressão sem perdas, serão expostas noções sobre a Teoria da informação de Claude Elwood Shannon.

É possível afirmar que a **Teoria da Informação** é a base matemática por trás das técnicas de compressão atuais. Shannon estipulou o conceito de auto-informação como a quatidade de informação que determinado evento carrega atrelada a probabilidade deste mesmo evento. Para Shannon, se $P(A)$ representa a probabilidade de um evento A ocorrer, então a auto-informação associada à A é dada por:

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

A base b referente ao logaritmo irá definir a unidade da informação, 2 para $bits$, e para $nats$ e 10 para $hartleys$. A representação logarítmica expressa a ideia sobre quando determinado evento possui uma alta probabilidade de acontecer, isso acrescenta pouca informação à análise. De grosso modo, é plausível assumirmos que pouca informação será necessária para representar um evento com alta probabilidade associada, enquanto um evento com baixa probabilidade detém uma grande auto-informação associada, logo muita informação será necessária para representá-lo. Outra propriedade importante da afirmação de Shannon é que a informação da ocorrência de dois eventos independentes é simplesmente a soma da informação obtida da ocorrência de cada um. Considere dois eventos independentes A e B. A auto-informação associada à ocorrência de ambos é:

$$i(AB) = i(A) + i(B). \tag{2.5}$$

Portanto, caso exista uma amostra de eventos independentes, podemos calcular a média do espaço amostral, ou em outras palavras, a entropia associada ao experimento.

A **entropia** pode ser definida como a medida do número médio de símbolos binários necessários para codificar a saída de determinada fonte. Por conseguinte, considere um conjunto de eventos independentes A_i, os quais são saídas de estipulado experimento $\mathcal{S}$, como:

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

Onde S é o espaço amostral. Então, caso o espaço amostral seja finito, pode-se escrever a entropia associada ao experimento explicitamente como:

$$H = \sum_i P(A_i) i(A_i) = -\sum_i P(A_i) \log_b P(A_i). \tag{2.7}$$

À vista disso, para compressões sem perdas, Shannon certificou que o melhor resultado esperado de uma compressão desse tipo é sempre a entropia da fonte para qualquer código que codifique essa fonte [16]. Por outro lado, como nem sempre é possível conhecer a entropia da fonte, a análise dos dados buscando estipular um modelo matemático que se adeque o máximo possível ao comportamento dos dados é de suma importância. Quanto melhor o modelo se adequar aos aspectos da realidade dos dados representados, maior a probabilidade da codificação apresentar um resultado satisfatório. Os Modelos Físicos, os Modelos Probabilisticos e os Modelos de Markov são exemplos de abordagens muito utilizadas até então [16].

Assim sendo, ao considerar elaborar um algoritmos para codificar uma fonte, na realidade, está se

considerando elaborar dois algoritmos distintos porém intimamente relacionados, o Encoder e o Decoder. O primeiro é o responsável pela codificação de fato. Esse, receberá uma entrada X e gerará uma saída Xc representando a entrada X com menos informação, enquanto o Decoder é o responsável por receber a mensagem codificada e decodificá-la de acordo. Nesse sentido, o Decoder receberá a mensagem codificada, Xc, e gerará a saída Y. No cenário em que Y é idêntico a X, têm-se compressor sem perdas, e quando não é, têm-se um compressor com perdas.

Tabela 2.1: Glossário de termos

Fonte	Gerador ou origem dos dados
Letra ou Símbolo	Unidade de informação ou dado
Alfabeto	Conjunto de letras existentes na Fonte
Mensagem	Sequência de letras
Código	Sequência binária que representa uma letra
Contexto	Relação probabilística entre letras

2.3 TÉCNICA DE COMPRESSÃO

De acordo com a nomenclatura da compressão de vídeo, *codecs* que comprimem cada frame de uma Point Cloud estática individualmente são chamados de *intra coders*, e são chamados de *inter coders* todos os outros *codecs* que buscam explorar a rendundância temporal de Point Clouds dinâmicas. Dito isso, ainda existem vários aspectos representados dentro de uma Point Cloud que podem ser comprimidos, geometria, cor, refletância e etc, dependendo da aplicação. Por mais que existam técnicas que também abordem a compressão desses outros aspectos, a técnica do presente estudo propõe um compressor *intra coder* para compressão de geometria de Point Clouds estáticas. Normalmente, ao falar de compressão de geometria, procura-se analisar a maneira com que os pontos de determinada estrutura estão distruibuídos a fim de encontrar espaços vazios e redudâncias entre os espaços preenchidos. Essa estratégia baseia-se na possibilidade de encontrar informação que pode ser removida do modelo 3D sem comprometer o seu conteúdo original. Dito isso, é pretendido representar a Point Cloud através de imagens binárias por meio de operações lógicas e usar técnicas clássicas de compressão de imagens binárias para enfim comprimir os dados de fato. Outro exemplo de algoritmo de compressão de geometria muito relevante para o estado-da-arte é o método da Octree [17], também muito debatido na literatura.

2.3.1 Octree

Nesse método a Point Cloud é dividida recursivamente em 8 cubos até que se atinja o nível dos voxels unitários, como ilustrado em 2.6. Uma forma mais simples de se compreender os conceitos por trás das Octrees, é analizar o modelo bidimensional, as chamadas Quadtrees. Analogamente, parte-se de um grande quadrado representando a imagem completa, esse é, então, dividido em quatro quadrados menores de tamanhos iguais entre si. Para cada quadrado ocupado atribui-se o bit 1, enquanto para quadrados vazios atribui-se o bit 0. Assim, pode-se realizar essas subdivisões recursivamente até que se alcance a dimensão unitária do pixel. A grande vantagem deste tipo de arquitetura é que grandes espaços vazios são codificados

com apenas a utilização de um bit, como pode ser visto no quadrante direto superior da segunda Quadtree na Figura 2.7

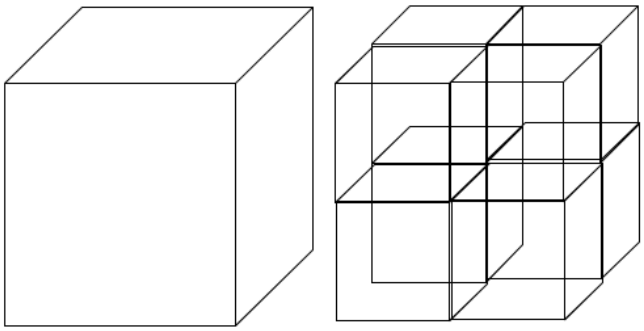

Figura 2.6: Cubo unitário divido em 8 sub-cubos. Figura cortesia de [1]

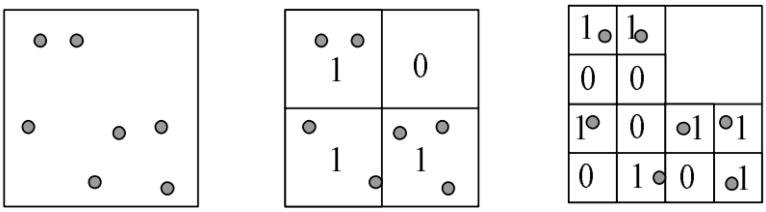

Figura 2.7: Exemplo de codificação do tipo quadtree. Figura cortesia de [4]

Sob essa perspectiva, a estrutura da decomposição de uma Octree é uma extensão 3D da decomposição de uma Quadtree [1], onde para cada cubo ocupado, é marcado 1 na decomposição da árvore e 0 quando é vazio. Através da Figura 2.8 torna-se evidente que a compressão se dá principalmente, pois da mesma forma que na Quadtree, os volumes vazios marcados com 0 são transmitidos utilizando apenas 1 bit. Assim, todas as subidivisões formarão uma sequência de bits que pode, sem perda de informação, reconstruir a imagem original. Uma estratégia similar ao modo como os volumes vazios são marcados será debatida no algoritmo proposto.

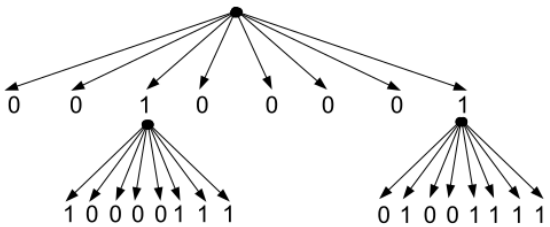

Figura 2.8: Processo de codificação da árvore de volume. Figura cortesia de [1]

A abordagem das Octrees é tão relevante no que diz respeito a codificação de Point Clouds devido ao fato de que além de garantir uma taxa de compressão considerável quando comparada aos outros métodos de compressão [18], são também capazes de modificar a resolução da imagem 3D a depender da taxa desejada, incrementando o número de níveis de profundidade da árvore, como ilustrado em 2.9.

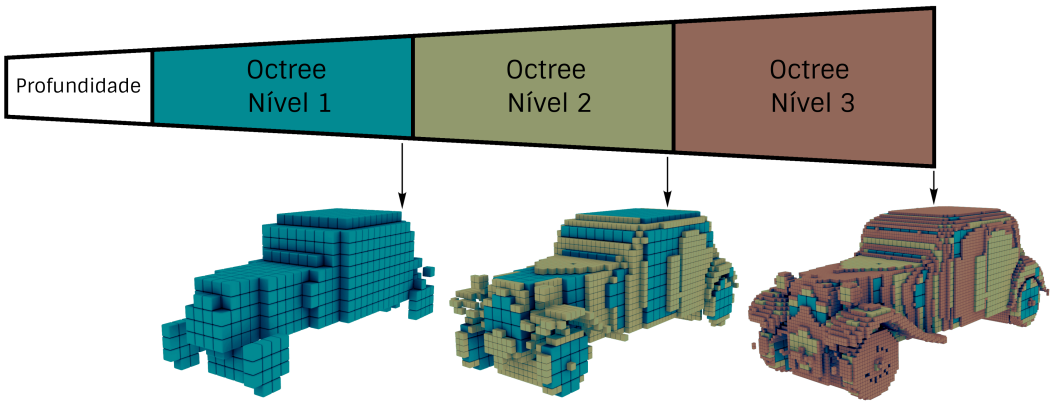

Figura 2.9: Níveis de resolução de uma estrutuda Octree. Figura adaptada de [5]

Como a cada subdivisão dos cubos em 8 cubos menores um novo byte é necessário, quanto maior a resolução, ou seja, menores os cubos, maior o número de bytes necessários para enviar a imagem. O modelo de octree permite que seja preestabelecido uma dimensão mínima dos cubos, diminuindo a resolução porém diminuindo a taxa de transmissão também. Essa escalabilidade é um grande diferencial da metodologia de octrees principalmente no que diz respeito à renderização de espaços tridimensionais. Pode ser feita uma renderização dinâmica na qual à medida que se aproxima de um objeto, mais bem definido ele se torna.

2.3.2 Silhuete 3D

A *decomposição em silhuetas* da Point Cloud consiste basicamente em fatiar e achatar a geometria 3D de uma Point Cloud N x N x N dada sobre determinado eixo, até alcançar o nível das fatias unitárias N x N x 1 que compõe a Point Cloud. Por razão de tratarmos unicamente da geometria dessas estruturas, teremos como informação apenas se as coordenadas são ocupadas ou não ocupadas. Nesse sentido, a Point Cloud é fatiada recursivamente em intervalos, ou seções. Uma seção será definida como o conjunto de fatias unitárias que constituem determinado intervalo. Dessa forma, no primeiro momento, a estrutura é dividida em duas seções menores. As duas seções menores são divididas novamente em 2 seções ainda menores. O processo de divisão é repetido até que a seção seja composta apenas por uma fatia ou seja encontrada uma seção vazia, similar ao que acontece no método da Octree, explicitado em 2.8. Por exemplo, se decidimos fatiar uma Point Cloud do tipo (x, y, z) = 8 x 8 x 8 sobre o eixo Z, primeiro iría se obter 2 seções de 8 x 8 x 4, em seguida ambas seriam dividídas novamente, então teríamos 4 seções do tipo 8 x 8 x 2, até finalmente alcançarmos as 8 fatias unitárias 8 x 8 x 1 ao longo do eixo Z, as quais sequencialmente, compõem a Point Cloud original. A Figura 2.10 busca ilustrar exatamente esse raciocínio no caso em que todas as seções possuem ao menos algum voxel ocupado.

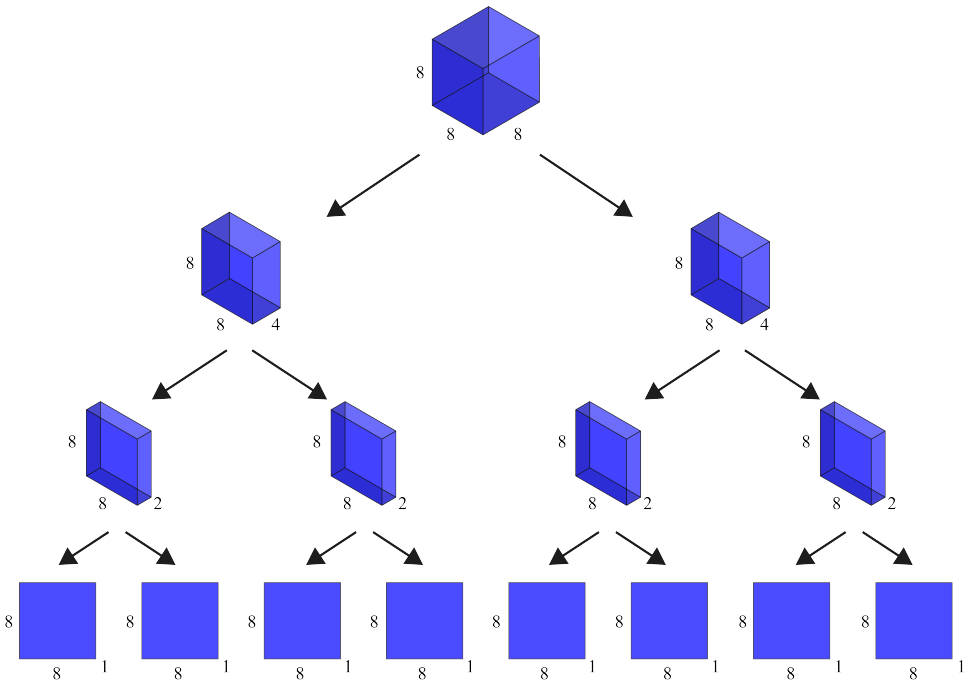

Figura 2.10: Fatiamento diádico de uma Point Cloud 8 x 8 x 8.

Entendido as premissas de como o processo do *fatiamento* é dirigido, agora podemos explicar como as silhuetas são obtidas. É realizado o chamado *achatamento*, o qual consiste em obter o somatório de todas as fatias unitárias presentes em cada seção através de operações lógicas **OU**. Todas as fatias ditas são, então, fundidas resultando em uma única imagem 2D N x N binária que representa uma estrutura similar ao que seria a silhueta daquela seção. Mediante esse processo, vamos tomar como exemplo uma Point Cloud 4 x 4 x 4 preenchida pelos seguintes voxels (x, y, z) - (1, 0, 3), (1, 1, 3), (2, 1, 3), (1, 1, 2), (1, 2, 2), (1, 1, 1), (2, 1, 1), (1, 2, 1), (1, 2, 0), (1, 3, 0), (2, 3, 0), (3, 3, 0). À vista disso, torna-se evidente que a estrutura é composta por 4 fatias unitárias 4 x 4 e, realizando o fatiamento sobre o eixo Z, teríamos como resultado as fatias listadas abaixo e ilustradas em Figura 2.11.

- Z = 3.

  (1, 0, 3), (1, 1, 3), (2, 1, 3).

- Z = 2.

  (1, 1, 2), (1, 2, 2).

- Z = 1.

  (1, 1, 1), (2, 1, 1), (1, 2, 1).

- Z = 0.

  (1, 2, 0), (1, 3, 0), (2, 3, 0), (3, 3, 0).

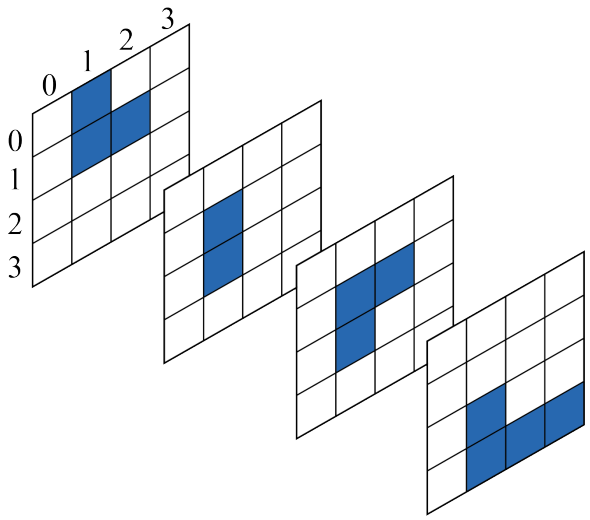

Figura 2.11: Exemplo de fatias unitárias 4x4 que compõe uma Point Cloud.

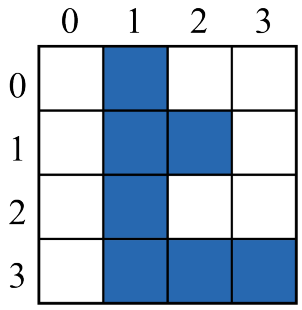

Figura 2.12: Resultado do achatamento de uma seção composta por 4 fatias.

No que diz respeito as seções vazias, no instante em que uma é encontrada, não há mais necessidade em dividi-la, pois isso significa que todas as fatias unitárias dentro dessa seção também são vazias. Podemos deduzir isso devido as operações **OU** realizadas.

Desde o princípio, ao dividir a Point Cloud em metades, estamos formando a estrutura de uma árvore binária que contém as seções achatadas conforme as prosseguimento das divisões. Como resultado final, teremos algo que chamamos de árvore binária de silhuetas, conforme revelado na Figura 2.13.

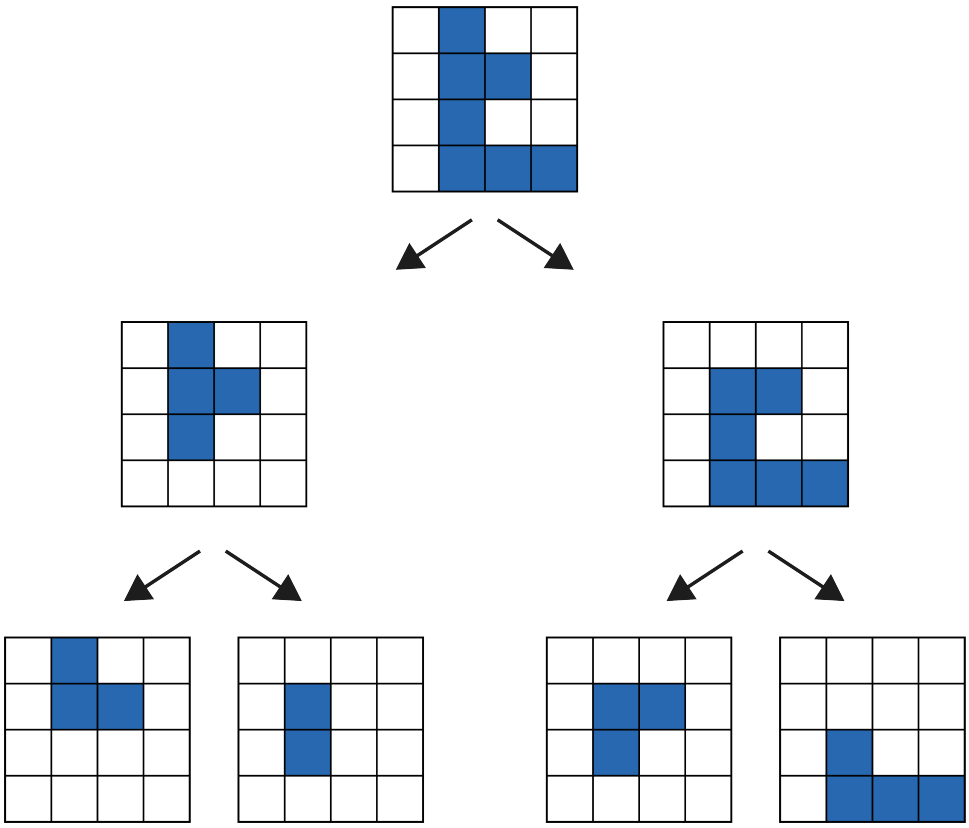

Figura 2.13: Árvore binária derivada da decomposição em silhuetas.

Se analisarmos a Figura 2.13 com mais ênfase, é possível perceber que podemos aproveitar o modo como a estrutura é organizada tomando vantagem sobre as semelhanças compartilhadas entre os nós de silhuetas vizinhos, mais especificamente sobre o nó pai e o outro nó gerado por esse mesmo pai, o nó irmão. É importante ressaltar que os nós simbolizam silhuetas, ou seja imagens binárias e que a decomposição não ocorre sobre o nó raiz, afinal este não apresenta nem nó pai, nem nó irmão, portanto este é enviado por completo. Sob esse víes, dado um nó Y_P da árvore e seus nós filhos, Y_E, nó filho à esquerda, e Y_D, nó filho a direita, é pretendido transmitir Y_E e Y_D, considerando que Y_C já foi transmitido. Então são realizados os seguintes passos:

1. Na transmissão de Y_E, dada a imagem pai Y_P como a máscara da decomposição, apenas os bits onde o valor de Y_P é 1, são enviados de Y_E.

2. Na transmissão de Y_D, dada a imagens Y_P e Y_E como máscaras da decomposição, apenas os bits onde o valor Y_E são 1, são enviados de Y_D.

A razão pela qual pode-se realizar a primeira decomposição, deve-se ao fato de que enxergamos a relação $Y_P = Y_E + Y_D$, onde a soma é uma operação lógica **OU**. A única possibilidade aonde Y_P é 0, é quando ambos Y_E e Y_D também são 0. Assim, considerando que Y_P já foi transmitido, é possível deduzir esses valores de ambos nós filhos. Portanto, só é preciso enviar os símbolos aonde Y_P é 1. Na segunda decomposição possuímos mais informação visto que Y_E já foi processado. À vista disso, considerando que Y_P e Y_E foram enviados, sabemos que $Y_P = Y_E + Y_D$, logo, se $Y_P = 1$ e $Y_E = 0$, significa que $1 = 0 + Y_D$

e consequentemente, $Y_D = 1$. Então, os símbolos onde o nó pai é 1 e no filho à esquerda é 0, no filho à direita deve ser 1. Sendo assim, na segunda decomposição, enviamos apenas os símbolos que o nó à esquerda são 1, pois conseguimos inferir os restantes.

Analisando os pixels azuis como pixels não-vazios, os brancos como vazios e o contorno em vermelho como os símbolos que devem ser transmitidos em cada imagem, como ilustrado na Figura 2.14. Pode-se observar que o contorno representa exatamente os símbolos ocupados das respectivas máscaras. Assim, sabendo que a raíz é sempre enviada completamente para a sequência de símbolos e que a decomposição das silhuetas seguem a travessia pré-ordem de árvore binária, a sequência de símbolos enviada de cada nó é dado por:

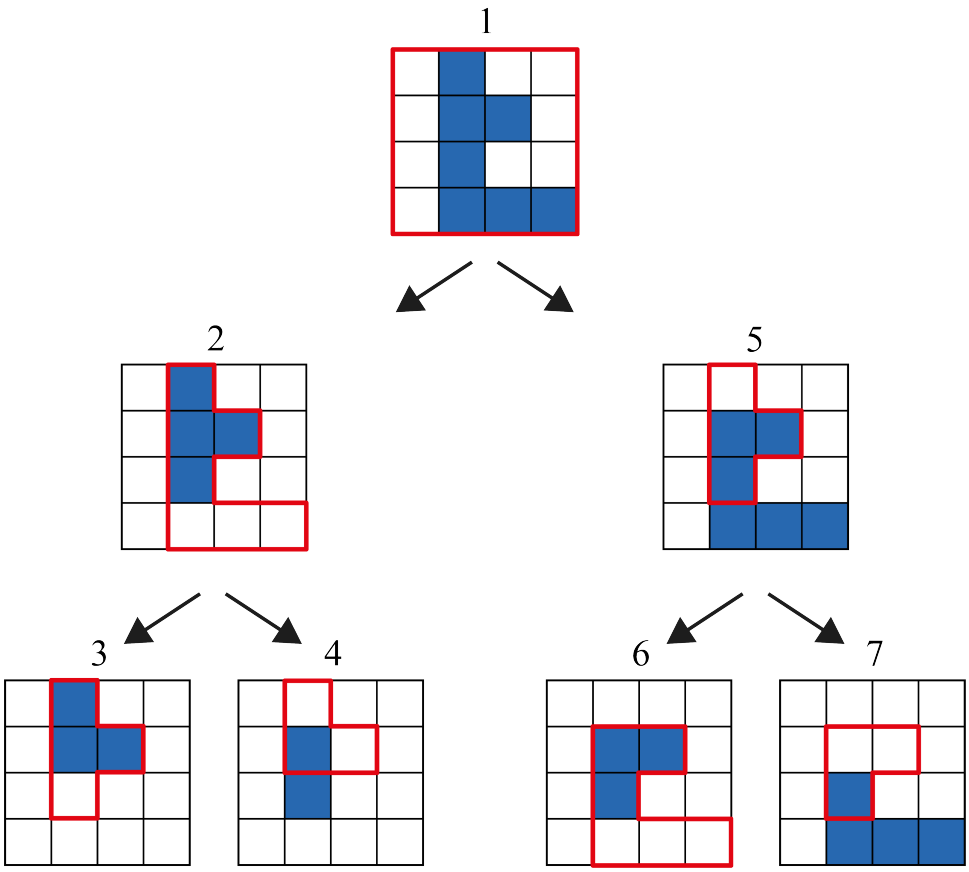

Figura 2.14: Compressão da decomposição em silhuetas.

1. 1: 0100011001000111
2. 2: 1111000
3. 3: 1110
4. 4: 010
5. 5: 0111
6. 6: 111000
7. 7: 001

Portanto, unindo cada sequência listada na ordem apresentada, a sequência de símbolos final ocupa 43 bits é dada por 01000110010001111111000111001001111111000001. Já é possível perceber algum ganho quando comparamos aos 64 bits necessários para enviar as 4 imagens 4 x 4. É importante ressaltar que o método se beneficia de Point Clouds maiores e mais esparsas, obtendo resultados ainda melhores, conforme apresentado no Capítulo 4.

A decodificação se revela trivial visto que a primeira silhueta é envianda completamente. Portanto é apenas necessário reconstruir a primeira imagem e então conhecendo a sequências de bits de cada etapa, pode-se reconstruir cada máscara e, consequentemente, toda a árvore de silhuetas. Ao fim, basta unir as silhuetas N X N X 1 presentes nos nós folhas na ordem correta para obter a Point Cloud original novamente.

2.3.3 Codificação Aritmética

Por mais que o resultado da compressão de geometria resultante da decomposição das silhuetas já se revele expressivo, esse resultado ainda era superado em desempenho quando comparado aos números dos principais codificadores do estado-da-arte. Nesse sentido, após a decomposição em silhuetas ocorrer e descartarmos todos os símbolos desnecessários para a recomposição da Point Cloud no futuro, agora aplicamos sobre a sequência de símbolos um dos principais codificadores aritméticos que possuímos atualmente, buscando alcançar uma taxa de compressão ainda melhor unindo as duas técnicas.

A técnica da *codificação aritmética* consiste em encontrar um identificador único dentro de um intervalo numérico que represente a mensagem a ser codificada utilizando as probabilidades de ocorrência de cada símbolo dentro da própria mensagem. Para isso, precisamos relacionar as probabilidades e os símbolos dentro da mensagem de maneira disjunta, ou seja, precisamos de uma função que mapeie variáveis aleatórias (os símbolos) à um dado intervalo único. Assim, o candidato perfeito para este mapeamento é a função de distribuição acumulada - CDF - que mapeia variáveis no intervalo $[0, 1)$ acumulando as probabilidades das variáveis dadas. Como existem infinitos números no intervalo entre zero e um, temos identificadores suficientes para qualquer mensagem que venha a ser codificada. A CDF origina um intervalo único $[F(x), F(x-1))$ para cada probabilidade $p(x)$ atrelada a um x_i pertencente a um alfabeto $\mathcal{A} = x_1, x_2, ..., x_i$, onde

$$F(x_i) = \sum_{1}^{i} p(x_i) \tag{2.8}$$

.

conforme ilustrado através dos gráficos presentes na Figura 2.15.

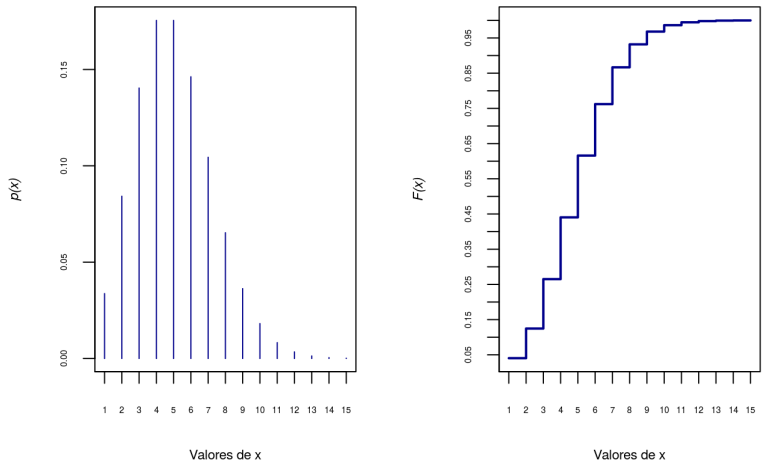

Figura 2.15: Para cada amostra de probabilidade $p(x)$ à esquerda, há um intervalo $[F(x), F(x-1))$ único relacionado à direita. Adaptado de [6].

Desse modo, para gerar o identificador único, parte-se do intervalo [0,1) e nele identifica-se o sub-intervalo ao qual corresponde o primeiro símbolo lido da mensagem. Para cada símbolo subsequente, subdivide-se o intervalo atual em sub-intervalos proporcionais às probabilidades da tabela de intervalos, e encontra-se novamente o intervalo que corresponde ao próximo símbolo, formando um processo iterativo, em que a cada iteração, menor é o intervalo que representa cada símbolo. Ao final do processo, teremos um intervalo que corresponde a probabilidade da ocorrência de todos os símbolos lidos na ordem correta. Então, basta enviar qualquer identificador possível dentro desse intervalo numérico que o decodificador conseguirá recuperar a mensagem original, basta este saber de antemão todos os símbolos, as probabilidades e a quantidade de símbolos na mensagem, que normalmente são enviados anteriormente na forma de *header*. A figura abaixo ilustra o caminho de divisões e subdivisões sucessiva dos intervalos.

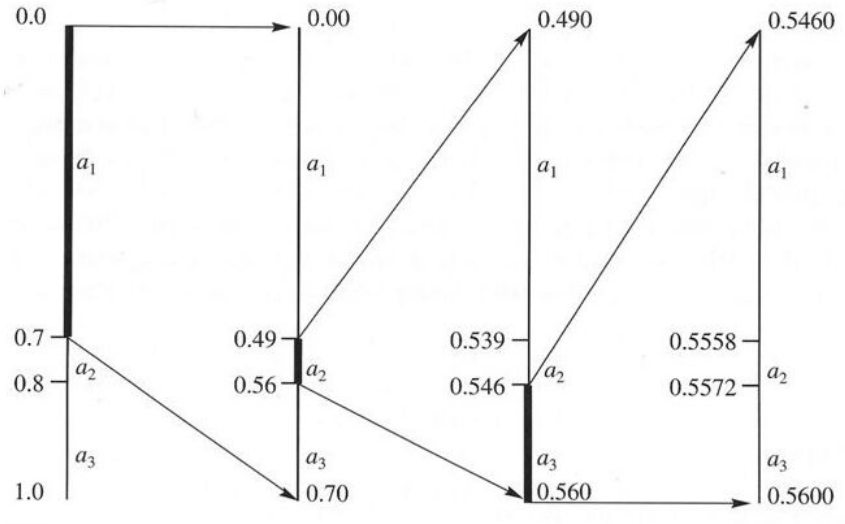

Figura 2.16: Processo de divisão dos intervalos na codificação aritmética.

No que diz respeito ao desempenho, de modo geral, codificadores aritméticos conseguem produzir uma saída próxima da entropia, $H(\mathcal{S})$, para qualquer conjunto de símbolos dados. Agrupando os símbolos em

blocos de m símbolos, temos que o limite superior do codificador aritmético é definido por

$$R \leq H(\mathcal{S}) + 2/m. \tag{2.9}$$

Aprimorando a ideia da codificação aritmética, vamos chegar ao conceito de *codificador aritmético com contexto*. Essa técnica segue exatamente a mesma estratégia do codificador aritmético clássico, porém se imaginou que poderia aproveitar as informações já adquiridas dos símbolos anteriores da mensagem como contexto para a codificação dos próximos símbolos, ajustando ainda mais as probabilidades e obtendo mais precisão, consequentemente mais compressão. Dessa forma, no cenário do codificador aritmético, dizemos que codificamos a mensagem com um contexto quando alteramos o modelo de probabilidades de acordo com o que já foi codificado. Por exemplo, considere a seguinte mensagem:

$$(x_1, x_3, x_3, x_2, x_1, x_3, x_3, x_3, x_3, x_1, x_1, x_2, x_2)$$

Se observarmos a frequência de x_3 nessa sequência, podemos perceber que x_3 acontece aproximadamente 46.15% das vezes, porém se já é conhecido o símbolo decodificado anteriormente, podemos utilizá-lo como contexto para adicionar mais precisão sobre a probabilidade das ocorrências do nosso modelo. Nessa sequência, por exemplo, x_3 ocorre uma vez após x_1, uma vez após x_2 e quatro vezes após o próprio x_3. Portanto, se é conhecido que o símbolo processado anteriormente foi x_3, a probabilidade de x_3 aumenta para 66.67%. Por conseguinte, como na codificação buscamos representar símbolos mais frequentes com menos bits e símbolos menos frequentes com mais bits, se a predição estiver correta, teremos encontrado quais símbolos são mais frequentes para cada contexto e, consequentemente, economizado bits.

Se analisarmos a ideia de contexto ainda mais profundamente, é possível perceber que estamos alterando a tabela de probabilidades sempre que preferimos um contexto em prol de outro durante o processamento dos símbolos, e consequentemente, alterando, também, o intervalo $[F(x) - F(x-1))$ associado a cada símbolo lido. Mesmo que não seja tão intuitivo, na codificação aritmética, pode-se alterar a tabela de probabilidades usada até mesmo em todas as iterações, se for conveniente fazê-lo. A única condição vigente para que isso ocorra sem erros, é o codificador e o decodificador estarem sincronizados e utilizarem a mesma n-ésima tabela de probabilidades na mesma n-ésima iteração. Note que a modelagem do contexto impacta diretamente o desempenho do algoritmo e não estamos limitados a utilizar apenas 1 bit de contexto, porém quanto mais bits, maiores serão as tabelas de contexto, produzindo 2^m possíveis contextos para m bits utilizados. O JBIG, um codificador de imagem binárias que inspirou a modelagem dos contextos no presente estudo [19], utiliza em torno de 10 à 16 bits de contexto e o presente estudo, em algumas tabelas chega, a utilizar 20 bits de contexto.

Com isso em mente, no exemplo acima, era assumido que as tabelas de probabilidades se encontravam disponíveis e computadas antes mesmo da codificação ou decodificação começarem de fato. Porém, na maioria dos casos, essa informação não está disponível para o algoritmo desde o princípio. Uma possível solução seria o próprio algoritmo construir a tabela de probabilidades durante sua execução e *adaptar* as probabilidades dos símbolos conforme o processamento progride. Para isso, como ilustrado na Tabela 2.2, poderíamos iniciliazar a tabela de probabilidades com os contadores de todos os símbolos em 1 e para cada símbolo codificado na n-ésima iteração, incrementar seu contador e o contador total de ocorrências. Dessa

maneira, teríamos acesso a probabilidade adaptada de cada símbolo após cada ocorrência em tempo de execução. A mesma estratégia é reproduzida no decodificador.

Tabela 2.2: Distruição de frequências dos símbolos para cada n-ésima iteração.

	n = 1	n = 2	n = 3	n = 4	n = 5		n = 14
x1	1	2	2	2	2		5
x2	1	1	1	1	2	...	4
x3	1	1	2	3	3		7
Total	3	4	5	6	7		16

Assim, é razoável pensar que a seleção do contexto atua como uma chave que seleciona um contexto dentro de uma gama de tabelas de probabilidade possíveis, enquanto a adaptação é responsável por atualizar estes mesmos contextos conforme os símbolos são processados.

Finalmente, pode-se introduzir o codificador escolhido para atuar sobre os bits resultantes da abordagem S3D no presente estudo, o **Codificador Aritmético Binário Adaptativo ao Contexto**, que une todos esses conceitos apresentados em um unico codificador. Pensado nisso, as silhuetas operam justamente através de representações binárias, logo torna-se factível operar com mensagens que possuem apenas dois tipos de símbolo. A principal vantagem de ter um modelo de probabilidade com apenas dois símbolos no alfabeto é que este modelo consiste na probabilidade de apenas de um dos símbolos, afinal, se quisermos descobrir a probabilidade do outro, precisamos meramente fazer 1 menos a probabilidade já conhecida. Por conseguinte, o requerimento de apenas um valor para representar o modelo de probabilidades permite que possamos expandir a tabela de probabilidades com várias combinações e, assim, diversos contextos para codificar nossa sequência, garantindo um método de compressão ainda melhor. O exemplo abaixo abordará a ideia do funcionamento do codificador em questão conforme os conceitos apresentados acima. Vamos supor que queiramos enviar a seguinte mensagem

1100011.

Ao inicializar as tabelas de contexto teremos algo como ilustrado abaixo. Mesmo que todos os símbolos estejam sendo representados a fim de ilustrar melhor o funcionamento do codificador, repare que, como dito anteriormente, não é necessário enviar as duas colunas dos símbolos, afinal é possível encontrar o valor de uma através da já conhecida fazendo 1 menos a probabilidade da primeira.

Tabela 2.3: Tabela de contextos inicializada.

		Símbolos		
		0	1	Total
Contexto	Nenhum	1	1	2
	0	1	1	2
	1	1	1	2

Dessa forma, durante a primeira iteração verificamos que nosso símbolo atual é 1 e não possui contexto pois não há símbolo anterior a ele. Note que esse é o único caso em que isso irá acontecer, consequentemente é o único caso em que utilizamos a tabela com nenhum contexto. Assim, como podemos inclusive descartar

essa tabela na próxima iteração, não precisamos atualizar a tabela de contextos nesse momento. Dando continuidade, se verificarmos na Tabela 2.3, veremos que a frequência registrada quando nosso símbolo atual é 1 e não há símbolo anterior, é de 1, como ilustrado em verde na Tabela 2.4. Portanto, para encontrar sua probabilidade apenas se faz sua frequência dividida pelo total de ocorrências daquele contexto. Se fizermos isso, iremos obter $1/2$ como resultado.

Tabela 2.4: Iteração 1.

símbolo anterior	símbolo atual	p(x) atual
Nenhum	1	1/2

		Símbolos		
		0	1	Total
Contexto	Nenhum	1	1	2
	0	1	1	2
	1	1	1	2

Após a primeira iteração, temos informação anterior e podemos começar a atualizar a tabela de contextos conforme os símbolos são processados. Na segunda iteração, o próximo símbolo, segundo bit da mensagem, também é 1, então a probabilidade que estamos procurando é a de quando o símbolo anterior e o símbolo atual são 1. Repetindo o processo anterior, vamos encontrar novamente $1/2$, porém dessa vez devemos atualizar nossa tabela de contextos incrementando tanto o contador do símbolo, quanto o contador do total de ocorrências daquele contexto.

Tabela 2.5: Iteração 2.

símbolo anterior	símbolo atual	p(x) atual
1	1	1/2

		Símbolos		
		0	1	Total
Contexto	0	1	1	2
	1	1	1	2

Tabela 2.6: Tabela de contextos na iteração 3.

		Símbolos		
		0	1	Total
Contexto	0	1	1	2
	1	1	2	3

Repetindo esse processo até o fim da mensagem teremos cada probabilidade usada durante as divisões subsequentes dos intervalos para encontrar um identificador que represente a mensagem. À vista disso, o desenvolvimento do raciocínio apresentado é ilustrado abaixo.

Tabela 2.7: Iteração 3.

Símbolo anterior	Símbolo atual	p(x) atual
1	0	1/3

		Símbolos		
		0	1	Total
Contexto	0	1	1	2
	1	1	2	3

Tabela 2.8: Iteração 4.

Símbolo anterior	Símbolo atual	p(x) atual
0	0	1/2

		Símbolos		
		0	1	Total
Contexto	0	1	1	2
	1	2	2	4

Tabela 2.9: Iteração 5.

Símbolo anterior	Símbolo atual	p(x) atual
0	0	2/3

		Símbolos		
		0	1	Total
Contexto	0	2	1	3
	1	2	2	4

Tabela 2.10: Iteração 6.

Símbolo anterior	Símbolo atual	p(x) atual
0	1	1/4

		Símbolos		
		0	1	Total
Contexto	0	3	1	4
	1	2	2	4

Tabela 2.11: Iteração 7.

Símbolo anterior	Símbolo atual	p(x) atual
1	1	2/4

		Símbolos		
		0	1	Total
Contexto	0	3	2	5
	1	2	2	4

Tabela 2.12: Tabela de contextos ao final da codificação.

		Símbolos		
		0	1	Total
Contexto	0	3	2	5
	1	2	3	5

As divisões matemáticas feitas pelo codificador aritmético estão ilustradas abaixo de acordo com as probabilidades geradas no exemplo em questão. Ainda é importante ressaltar que o processo de divisões dos intervalos e o processamento dos contextos estão entrelaçados, ou seja, os dois acontecem a cada iteração.

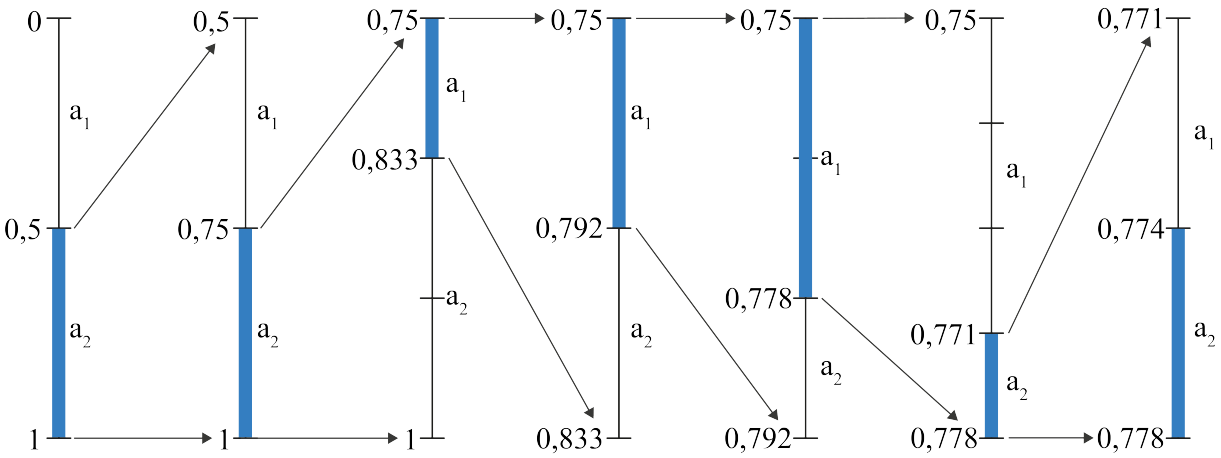

Figura 2.17: Intervalos provenientes da codificação aritmética.

Dessa forma, apenas é preciso enviar qualquer fração entre 0,778 e 0,774 que o decodificador conseguirá recuperar a mensagem.

Integrando o resultado da abordagem do SD3 com a codificação do codificador de entropia, é factível dizer que a estratégia da compressão de geometria apresentada, de maneira geral, é dirigida da seguinte forma: Dada uma Point Cloud, é realizada a decomposição desta em silhuetas e através da silhuetas torna-se possível identificar quais são os únicos símbolos necessários para a reconstrução da Point Cloud, descartando a outra parcela. Assim, é realizada a leitura de cada símbolo que a abordagem S3D relatar ser necessário e uma probabilidade é entregue pela tabela de contextos associado a esse símbolo. A tabela de contextos é atualizada e o codificador inclui esse símbolo dentro da codificação. Para o caso dos símbolos presentes na silhueta porém descartados por serem desnecessários, estes ainda são usados na atualização da tabela de contextos, no entanto não são enviados ao codificador. O raciocínio é ilustrado no diagrama abaixo.

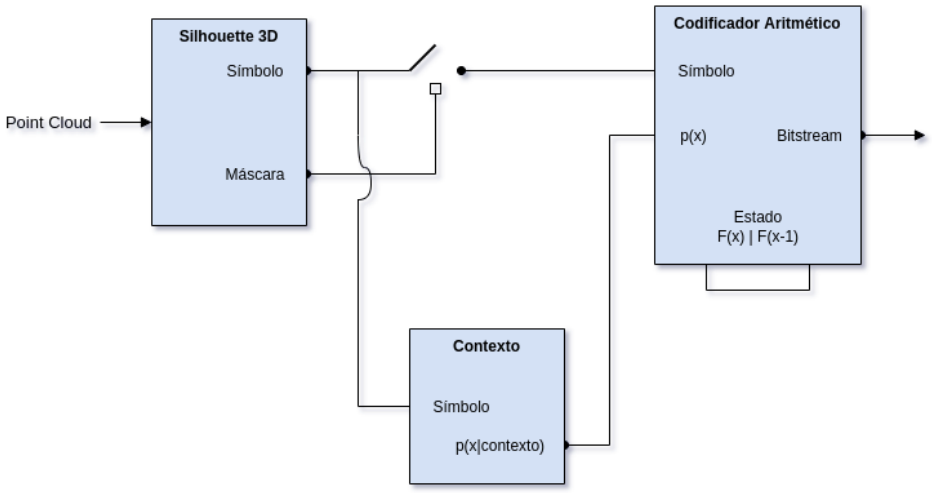

Figura 2.18: Roteiro da técnica de compressão S3D apresentada.

3 MÉTODO PROPOSTO

A migração de um software corresponde a uma forma de modernização, na qual uma aplicação existente é movida para um novo ambiente que oferece algum benefício desejado (manutenção, novas funcionalidades ou desempenho). Ademais, toda a refatoração do código acarreta na reanálise deste e, consequentemente, uma melhor organização. Dessa maneira, visando um código mais robusto, com melhor gerenciamento de memória, mais rápido e, principalmente, que possa competir com os algoritmos estado-da-arte enviados ao grupo de trabalho MPEG, uma estratégia de re-engenharia foi escolhida para o cumprimento desta tarefa. A estratégia proposta é comumente utilizada para a migração de aplicações de maneira sistematizada [20][21]. Para o presente estudo, foi realizada uma abordagem adaptada de [21] que consiste em 3 etapas. Primeiramente, utilizamos de engenharia reversa para obter uma representação em alto nível de abstração do sistema atual, como representado na Figura 2.18. Em um segundo momento, essa representação é analisada e administrada para se enquadar na arquitetura da nova linguagem. Finalmente, na terceira etapa, idealizamos uma representação reestruturada para o novo sistema e aplicamos à nova linguagem.

Por mais que o MATLAB seja uma excelente ferramenta para prototipagem de projeto e operações matriciais, este ainda não entrega uma liberdade tão grande quanto C++ ao falarmos de administração de memória. Além disso, por se tratar de uma linguagem interpretada, essa necessita primeiro que o código fonte seja colocado na memória para que então o interpretador leia-o linha por linha e o traduza em linguagem de máquina. Por outro lado, um programa compilado, normalmente, é mais rápido devido ao fato de que é otimizado e executado imediatamente, sem a necessidade de um interpretador. No contexto de sistemas embarcados, sabe-se que codificadores são frequentemente empregados por circuitos integrados e por isso, normalmente temos uma velocidade limitada de processamento e memória. Nesse cenário, é interessante termos controle sobre exatamente o que a CPU processa ou deixa de processar. Essa característica está presente em linguagens de programação como C++.

Entendido as motivações em transpor o código e as vantagens teóricas da linguagem C++ perante o MATLAB para o presente estudo, podemos debater as características do projeto legado para então compreender seu funcionamento e começar a migração. O projeto em MATLAB estava organizado da seguinte forma:

```
project
|---ArithmeticCoding
|---Bitstream
|---Decoder
|---Encoder
|---PlyTools
|---Structs
|---Utils
```

Cada pasta deste projeto corresponde a um módulo. Dentro de cada pasta, existem os arquivos associados

que implementam funções específicas ou estruturas de dados pertinentes a seu respectivo módulo. Essa abordagem, sob uma visão macro, é interessante visto que garante uma organização a nível de pastas e minimiza a complexidade dos arquivos de código ao menor nível possível, uma função por arquivo. Em contrapartida, quando um módulo apresenta muitas funções, a compreensão da relação entre estas pode se tornar mais desafiadora, posto que seu projeto pode acabar ficando com muitos arquivos. Além disso, no código, as estruturas de dados e as funções relativas a essas não são encapsuladas, isso as torna vulneráveis durante o decorrimento do software, pois podem ser acessadas e modificadas indevidamente por módulos externos, dificultando a manuntenção do código. Por exemplo, a dimensão da nuvem de pontos não deveria poder ser modificada pelo codificador aritmético, já que não faz sentido o codificador alterar algum dado da estrutura original. O tipo de legibilidade que o encapsulamento trás ao código é algo extremamente desejável, principalmente para projetos com vários desenvolvedores e rotatividade de membros.

Aliado às decisões pertinentes à linguagem escolhida, foi estabelecido que teríamos o Gitflow [22] como guia para controlar as submissões de pedaços de código. O Gitflow consiste em um modelo de ramificação o qual pretende garantir que o ramo principal do código sempre esteja livre de erros. Para isso, ramos auxiliares são criados de acordo com o desenvolvimento de novos requisistos, e enquanto um requisito não estiver completo e apurado, este não pode ser acoplado ao ramo principal. Dessa forma, podemos assegurar que o ramo principal irá contér todo código já testado e versionado, enquanto todo fluxo de trabalho ocorre paralelamente nos ramos de desenvolvimento auxiliares antes destes serem fundidos com o ramo principal. Na figura abaixo, esse raciocínio é ilustrado onde o primeiro ramo, na leitura de cima para baixo, é o principal e os derivados dele são ramos de requisitos em desenvolvimento. Cada esfera representa uma submissão, então note que o desenvolvimento de um requisito continua no ramo auxiliar por várias submissões antes de ser acoplado ao principal.

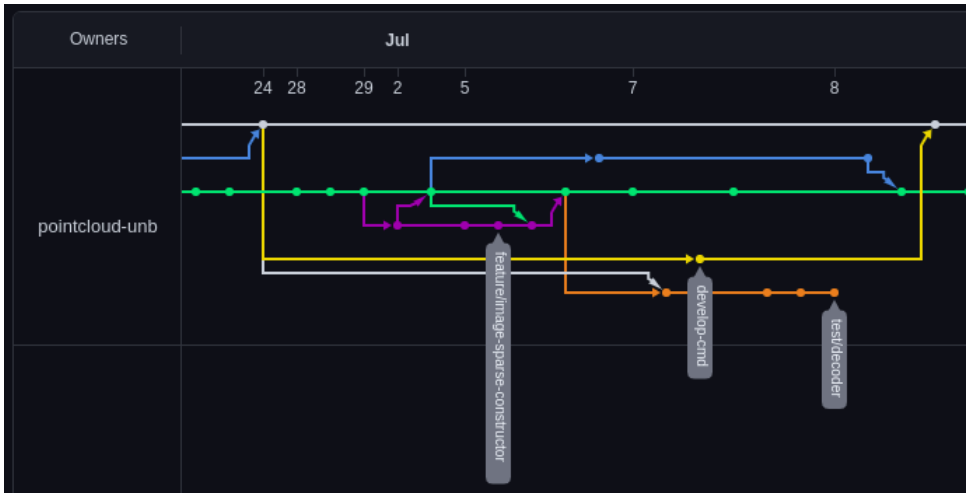

Figura 3.1: Rede de fluxo do repositório Github do projeto apresentado no mês de Julho e Agosto de 2021.

Assim, no contexto do C++, poderíamos optar pela abordagem de classes para organizar o projeto e ainda solucionar os problemas citados acima. Cada módulo ou combinação destes foi então enxergado como uma classe, restrigindo suas funções a métodos e cada informação pertinente a esses, a atributos. Visto que o projeto possuiria certo nível de complexidade, com múltiplos arquivos, diversas bibliotecas e seria alvo de vários desenvolvedores, foi optado incluir um arquivo Makefile para auxiliar no processo de

compilação do projeto. O Makefile e a organização de arquivos do projeto foram pensado para executar em todo tipo de sistema operacional em virtude de que na equipe haviam desenvolvedores em macOS (Apple), Windows (Microsoft) e Ubuntu (Linux). Sob esse viés, o projeto foi organizado da seguinte forma:

```
project
|---doc
|---include
|---legacy
|---src
|---test
|
| makefile
```

Todos os arquivos pertinentes à documentação do código foram alocados na pasta `doc`, os arquivos de cabeçalho na pasta `include`, os arquivos que foram implementados porém retirados do projeto principal, mas ainda detém algum valor foram alocados na pasta `legacy`, os arquivos fontes na pasta `src`, os arquivos objeto gerados pelo compilador na pasta `obj` e todos os arquivos relacionados aos testes unitários, explicados na Seção 3.2, na pasta `test`.

No que diz respeito à padronização de escrita do código do projeto, este seguiu as regras de nomenclatura do Google C++ Style Standard [23]. A ideia por trás de utilizar um padrão de codificação é atribuir clareza e consistência ao código, de maneira que até mesmo um leitor externo ao projeto tenha condições de identificar as estruturas do código corretamente. Dito isso, o padrão da Google incentiva nomes auto-explicativos, assim como o desuso de abraviações. De modo geral a descritividade de um nome deve condizer com o escopo de atuação do elemento associado a esse nome. Por exemplo, x pode ser um nome suficiente para uma variável dentro de uma função simples de 5 linhas, porém dentro do escopo de uma classe inteira, torna-se um nome vago. O guia do Google também estabelece regras mais objetivas quanto à nomeação de componentes do código [23].

Para nomear especificamente variáveis, incluindo parâmetros de funções e atributos, os nomes são sempre minúsculos e haverá underscores entre as palavras. No caso de atributos, estes devem possuir um underscore adicional ao final.

```
// Atributo da classe ImageRaster que representa o numero de pixels
// ocupados na imagem
int occupied_pixels_quantity_;
// Variavel auxiliar para armazenar o valor de um pixel
int value;
```

Namespaces são completamente minúsculos, com palavras separadas por underscores. Os nomes dos principais namespaces do projeto devem ser baseados no nome do próprio projeto.

```
// Principal namespace do projeto, Geometry-based Point Cloud Compression
namespace gpcc;
```

Funções e classes devem começar com e ter uma letra maiúscula para cada nova palavra no nome escolhido para a função.

```
// Classe abstrada IImage
class IImage;
// Funcao que verifica a existencia de um pixel dado de um objeto imagem
virtual bool PixelPresent(pix_t x0, pix_t y0) = 0;
```

Em relação a nomenclatura dos arquivos código do projeto, o padrão da Google exige apenas que os arquivos código devem ser sempre minúsculos e podem incluir underscores ou dashes. Portanto, foi estabelecido para o presente estudo o seguinte modelo: prefixo_nome_do_arquivo, onde o prefixo é uma sigla de duas letras que indica o módulo, o nome do arquivo será o nome relativo ao arquivo e o que ele representa de fato. Por exemplo, `ac_bac_encoder.cpp`, onde o prefixo ac remete ao módulo ArithmeticCoding, bac_encoder, significa Codificador Aritmético Binário.

```
include
| ac_bac_decoder.h
| ac_bac_encoder.h
| ac_bitstream.h
| ac_context.h
| . . .

src
| ac_bac_decoder.cpp
| ac_bac_encoder.cpp
| ac_bitstream.cpp
| . . .
```

Ao fim, foram elencados 5 módulos. O arithmetic coder (ac), responsável pela implementação do Codificador Aritmético Binário mencionado anteriormente. O módulo estrutura de dados (ds), resposável pela implementação de todas as estruturas de dados elaboradas, como imagens, point clouds, pixels e voxels. O módulo Encoder (en), responsável por toda a decomposição em silhuetas descrita, desde o fatiamento, o achatamento, a codificação através das máscaras, a análise dos nós da árvore e atualização do contexto. Por último, o módulo configuração (cf) é responsável pela definição dos parâmetros do arquivo de entrada usados na execução do algoritmo, como endereço das Point Clouds alvos. Na implementação original em MATLAB, por conta da sintaxe da linguagem, a maiora dos dados estavam sendo manuseados simplesmente como matrizes cruas ou *structs*. Portanto, foram implementadas novas estruturas de dados através da Standard Template Library (STL), buscando atingir o máximo de eficiência em termos de memória e tempo de execução. Ademais, é válido ressaltar que as estruturas de dados do presente estudo foram pen-

sadas para ser uma potencial API para projetos futuros que englobem o mesmo tema e então, permitir que outros usuários também utilizem das mesmas estruturas de dados e suas implementações.

3.1 CLASSES

Sob esse viés, dentro das estruturas implementadas, **Pixel e Voxel** representam as classes mais básicas de toda a implementação. Por outro lado, estas são intensamente utilizadas, visto que executam operações fundamentais para o desencadear do codificador. Operações algébricas de soma, subtração, multiplicação, divisão e operações boleanas como maior, menor, maior ou igual e menor ou igual são sobrescritas por meio da propriedade de *overload*. O recurso consiste, no contexto de linguagens de programação, em um artifício que permite a existência de vários métodos de mesmo nome, contanto que tenham assinaturas levemente diferentes, ou seja, variando em número, tipo de argumentos, valor de retorno e até variáveis diferentes. Quando um operador *overloaded* surge em uma expressão e pelo menos um de seus operandos tem um tipo de classe ou um tipo de enumeração, a resolução de overload é usada pelo compilador para determinar a função definida pelo usuário a ser chamada pela função com assinatura correspondente. Assim, uma quantidade considerável de código é poupada ao utilizar operações definidas pelos desenvolvedores em conjunto com as estruturas pré-definidas na biblioteca da linguagem.

```
/// Arithmetic operators ///
// Pixel + Pixel
template <typename U>
friend Pixel<typename std::common_type<T, U>::type>
operator+(const Pixel &lhs, const typename gpcc::Pixel<U> &rhs)
{
  return Pixel<typename std::common_type<T, U>::type>(lhs[0] + rhs[0],
   lhs[1] + rhs[1]);
}
```

Figura 3.2: Implementação de operação de soma entre objetos pixel.

À vista disso, foram criadas mais duas estruturas, Point Cloud e IImage, a partir das classes esclarecidas acima. Assim, podemos beneficiar as novas classes com todas as operações elementares de Voxel e Pixel já citadas.

A classe **Point Cloud** é composta por um `std::set` de voxels e tem como objetivo representar apenas os voxels ocupados da Point Cloud de fato. A escolha dessa implementação foi influenciada pelo motivo de que os dados da Point Cloud, no algoritmo S3D, não são tão manipulados durante a codificação, sendo requeridos apenas para obter as silhuetas. Uma vez que são conhecidos os voxels ocupados, temos informação suficiente para gerar as imagens binárias que representam as silhuetas, fazendo com que os voxels vazios não tenham tanta importância para o algoritmo. Dessa maneira, é possível poupar memória em tempo de execução, visto que a estrutura de uma Point Cloud tem três dimensões e quando geradas completamentes, revelam-se muito custosas para serem mantidas. Os voxels ocupados são então armazenados e mantidos ordenados a fim de facilitar a decomposição e recomposição da Point Cloud futuramente. As ações, como inserção, remoção e busca, feitas através da estrutura `std::set` tem com-

plexidade $O(log(n))$. Podemos destacar dois importantes métodos da classe Point Cloud, presentes no processo de obtenção das silhuetas para a codificação.

1. `SilhouetteFromPointCloud (int slice_start, int slice_stop, Axis axis)`
   A partir de um determinado eixo, fatia a Point Cloud em um determinado intervalo e, dessa fatia, projeta uma silhueta equivalente ao longo do mesmo eixo. Complexidade $O(n\ log_n)$.

2. `RecoverVoxelsFromSilhouette (Axis axis, int coordinate, ImageRaster* image)`
   Em um determinado eixo e coordenada neste eixo, insere as silhuetas de uma imagem. Complexidade $O(n\ log_n)$.

A classe **IImage** é uma classe abstrata e foi elaborada com o objetivo de representar as imagens binárias obtidas durante o fatiamento da Point Cloud. Naturalmente, essa classe age como super classe para classes derivadas concretas, essas são ImageSparse e ImageRaster. Cada uma das implementações derivadas de IImage é estruturada de maneira específica visando obter mais desempenho de execução. Em certos momentos, é necessário realizar incontáveis acessos aos Pixels da fatia gerada, como quando estamos calculando a tabela de contextos do codificador. Nesses momentos, é interessante minimizarmos custo de acesso, então idealizamos uma estrutura matricial do tipo `vector<vector<uint8_t>>` que apresenta complexidade de acesso $O(k)$, a **ImageRaster**. Em contrapartida, em determinados momentos, nos interessamos apenas pelos pixels ocupados da fatia, como no caso das máscaras. Portanto, não seria necessário representar a imagem completa em memória. Seguindo a mesma estratégia adotada na idelização da classe Point Cloud, podemos armazenar apenas os pixels ocupados por meio da estrutura `std::set`, ganhando em memória porém perdendo em tempo de acesso. Assim, foi implementado a **ImageSparse**. O relacionamento destes é explicitado pelo diagrama de classes em 3.3.

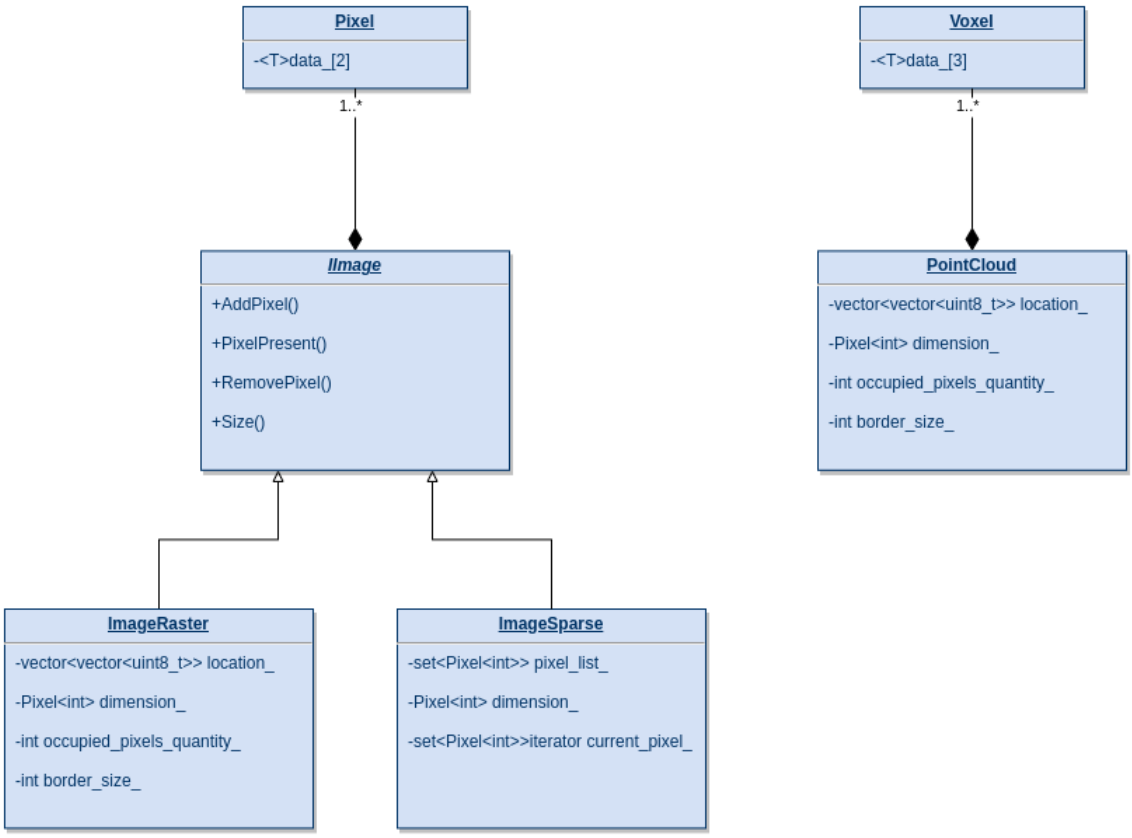

Figura 3.3: Diagrama de classes das estruturas de dados abordadas.

3.2 TESTES AUTOMATIZADOS

Dada as contribuições apresentadas ao projeto até então, enxergou-se interessante desenvolver testes unitários automatizados a fim de minimizar o risco e garantir manutenção do código. Testes unitários auxiliam a assegurar que cada requisito ou componente da aplicação funciona como deveriam [24]. Para isso, estes buscam validar o menor pedaço de código que pode ser logicamente isolado e testado, identificando erros simples antes destes escalonarem e se tornarem erros mais complexos. Dito isso, etapas de teste podem ser realizadas manualmente ou serem automatizadas. Na abordagem manual, o testador assume a função de um usuário final executando o software em teste para verificar seu comportamento e encontrar quaisquer possíveis defeitos. Sob outra perspectiva, em testes automatizados, usando certas ferramentas de teste, scripts de código de teste são desenvolvidos e executados, sem a intervenção de testadores humanos, para avaliar o comportamento do software. Se for planejado e implementado corretamente, a avaliação de software automatizada pode fornecer vários benefícios quando comparado ao teste manual [25]. Naturalmente, no contexto da necessidade de se recorrer à automação de testes, as ferramentas, ou *frameworks*, de automação ocupam um papel de protagonismo, pois são as grandes responsáveis pela execução desse tipo de tarefa. Avaliando a necessidade do projeto em questão, foram elecandas características desejáveis sobre a framework que viria ser utilizada a seguir.

1. Boa documentação,
2. Compatilidade com diversos sistemas operacionais,
3. Testes independentes e repetíveis,
4. Testes organizáveis,
5. Quando um teste falhar, este deve prover o máximo de informação sobre o problema possível.

Dessa forma, o framework de testes da Google, **Google Test (GTest)**, foi escolhida. Por possuir seu próprio arquivo *make* e suas próprias bibliotecas, este atua como um subprojeto isolado na pasta `test` dentro do presente trabalho. Visto que as novas estruturas de dados foram responsáveis por grande parte das alterações feitas no código legado durante a migração, as classes Pixel e Voxel foram exaustivamente testadas. Cada teste consistia em uma afirmação que poderia ser bem sucedida, uma falha não-fatal ou uma falha fatal. No caso de uma falha fatal acontecer, o programa de teste era interrompido, caso contrário prosseguia normalmente. A sintaxe para cada teste unitário é expressa ppor `TEST(TestSuit, UnitTest)`, onde `TestSuit` indica a categoria do teste a ser definido e `UnitTest` representa o teste específico a ser realizado, como ilustrado na Figura 3.4.

```
TEST(PixelConstructorsTest, XY) {
    /* Testing commom integers */
    gpcc::Pixel<uint32_t> p_aux1(0, 1);
    gpcc::Pixel<uint32_t> p_aux2(2016, 2022);
    /* Testing 32bit max range */
    gpcc::Pixel<uint32_t> p_aux3(4294967295, 32);
    /* Testing double || floats would acuse
    type error (so types are checked)*/
    gpcc::Pixel<double> p_aux4(3.3, 3.1415);
    gpcc::Pixel<double> p_aux5(0.923123, 8);


    ASSERT_EQ(0, p_aux1.x());
    ASSERT_EQ(1, p_aux1.y());

    ASSERT_EQ(2016, p_aux2.x());
    ASSERT_EQ(2022, p_aux2.y());

    ASSERT_EQ(4294967295, p_aux3.x());
    ASSERT_EQ(32, p_aux3.y());

    ASSERT_EQ(3.3, p_aux4.x());
    ASSERT_EQ(3.1415, p_aux4.y());

    ASSERT_EQ(0.923123, p_aux5.x());
    ASSERT_EQ(8, p_aux5.y());
}
```

Figura 3.4: Definição de teste unitário XY da categória PixelConstructorsTest.

A impressão de relatórios de falha foi uma característica da framework da Google bem explorada durante o desenvolvimento do projeto. A possibilidade de customizar mensagens de erro para cada teste, podendo

não só examinar melhor o teste falhado, mas também examinar efeitos colaterais sobre que não necessariamente são o foco do teste porém ainda apresentam importância para o desencadear da função testada, como ilustrado abaixo.

```
/* Expect no side effects (linking problem) */
EXPECT_EQ(old_x, p_aux3.x()) << "The sum on p1 changed the data inside p3";
EXPECT_EQ(old_y, p_aux3.y()) << "The sum on p1 changed the data inside p3";
```

Figura 3.5: Definição relatório de falha.

```
[       OK ] PixelConstructorsTest.Copy (0 ms)
[----------] 4 tests from PixelConstructorsTest (0 ms total)

[----------] 1 test from PixelIndexingTest
[ RUN      ] PixelIndexingTest.Indexing
[       OK ] PixelIndexingTest.Indexing (0 ms)
[----------] 1 test from PixelIndexingTest (0 ms total)

[----------] 11 tests from PixelOperatorsTest
[ RUN      ] PixelOperatorsTest.PlusEqualInt
/home/estevamalbuquerque/Documentos/UnB/TCC1/Github/geometryCoderCpp/test/PixelTest.cpp:150: Failure
Expected equality of these values:
  old_x
    Which is: 2
  p_aux3.x()
    Which is: 1
The sum on p1 changed the data inside p3
[  FAILED  ] PixelOperatorsTest.PlusEqualInt (0 ms)
[ RUN      ] PixelOperatorsTest.PlusEqualFloat
[       OK ] PixelOperatorsTest.PlusEqualFloat (0 ms)
[ RUN      ] PixelOperatorsTest.MinusEqualInt
[       OK ] PixelOperatorsTest.MinusEqualInt (0 ms)
[ RUN      ] PixelOperatorsTest.MinusEqualFloat
[       OK ] PixelOperatorsTest.MinusEqualFloat (0 ms)
[ RUN      ] PixelOperatorsTest.ShiftInt
[       OK ] PixelOperatorsTest.ShiftInt (0 ms)
[ RUN      ] PixelOperatorsTest.TimesEqualInt
[       OK ] PixelOperatorsTest.TimesEqualInt (0 ms)
```

Figura 3.6: Relatório de falha após erro induzido.

À vista disso, a esmagadora maioria dos testes foram elaborados em busca de assertir a funcionalidade de cada atividade ali testada. Todas as operações acobertadas pelas classes Pixel e Voxel foram testadas, desde todos os seus tipos de construtores até seus operadores relacionais para cada operação implementada. Na Figura 3.4, novamente, pode-se ver a descrição do teste unitário para um dos construtores da classe Pixel, utilizando duas coordenadas como parâmetros. Foram elencados, então, dois cenários no atual exemplo, onde no primeiro cenário são passados dois números inteiros e esperamos que os objetos criados aloquem corretamente seus valores, dois inteiros sem sinal, de 32 bits. Além disso, um caso especial no cenário de inteiros foi elencado, visando verificar o alcance máximo de uma variável daquele tipo durante seu instanciamento e operações. No segundo cenário, mesmo que não previsto pelo atual projeto, porém como é desejado expandir as classes aqui apresentadas para trabalhos futuros, também foram testadas unidades decimais.

Isso posto, foram pensados em ao menos dois tipos de cenários diferentes, para cada teste, porém coerentes com a sua categoria, e para cada cenário ao menos dois exemplos distintos também foram elaborados. No final obtivemos 50 casos de teste unitários em 10 categorias, somando no total em torno de 214 exemplos de teste.

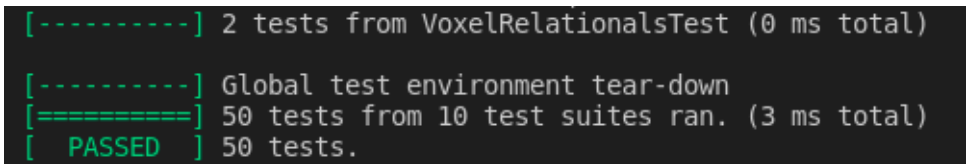

Figura 3.7: Sucesso em todos os testes.

3.3 DOCUMENTAÇÃO

Buscando corroborar com a legibilidade do código e incentivar o uso da API proposta, toda a implementação foi feita seguindo o protocolo do Doxygen de documentação através de comentários. O doxygen é um programa que gera a documentação de um programa a partir da análise dos comentários feitos ao longo do código e estruturados de maneira prevista pelo protocolo. Dessa forma, é possível produzir uma variedade de documentos que descrevem os componentes do código e suas relações automaticamente. Como realizar comentários em seu código de forma que a última versão corrente do Doxygen os incorpore na documentação gerada pode ser encontrada em [26]. É possível ver exemplos do que pode ser originado espontaneamente, através da abordagem em questão, nas figuras abaixo.

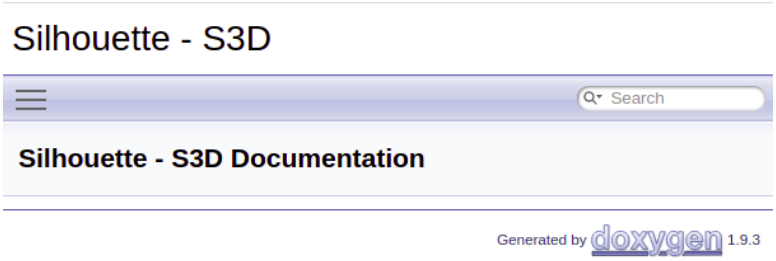

Figura 3.8: Página inicial da documentação HTML gerada pelo Doxygen.

Figura 3.9: Relação de construtores e destrutores da classe Image Sparse gerada pelo Doxygen.

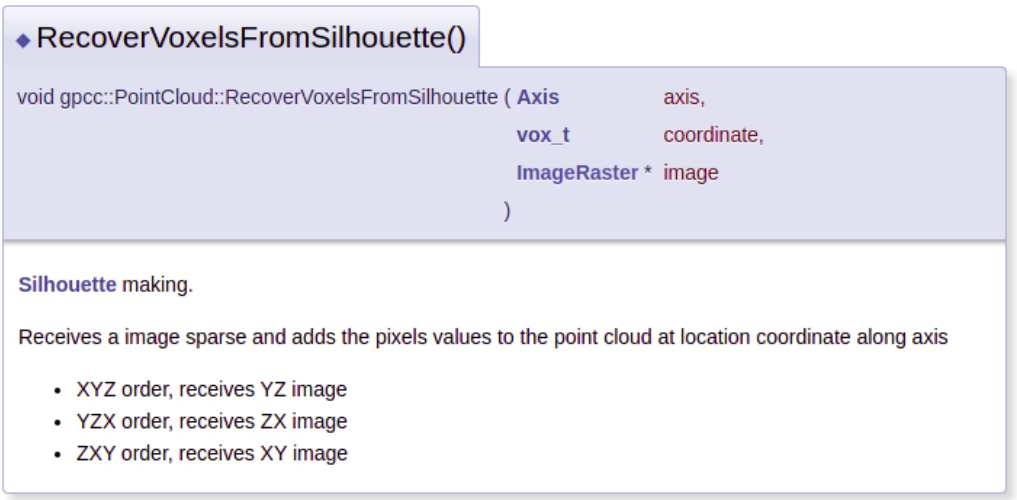

Figura 3.10: Descrição da função RecoverVoxelsFromSilhouette gerada pelo Doxygen.

CONCLUSÕES

Em síntese, foram discorridos os conceitos do porquê a migração do algoritmo proposto se fez necessária e como iria se estabelecer a organização do projeto para concrentizá-la. Atrelado a isso, foram

elucidadas todas as abstrações de estruturas de dados implementadas através da STD Library da linguagem de programação C++ na API proposta neste capítulo. Testes unitários também foram elaborados a fim de assegurar que cada requisito ou componente da aplicação funciona como deveria, identificando erros simples antes destes escalonarem e se tornarem erros mais complexos que poderiam comprometer a corretude do codificador. Dessa forma, para que toda informação também esteja presente dentro do código, toda a implementação foi feita seguindo o protocolo do Doxygen de documentação através de comentários. O código pode ser encontrado em: [27].

4 RESULTADOS

Neste capítulo será debatido o quanto das expectativas iniciais do estudo foram concretizadas através da comparação dos resultados dos algoritmos citados nesse trabalho.

4.1 RESULTADOS OBTIDOS

A avaliação dos resultados do presente estudo será feita através da comparação das medidas de taxa de compressão e tempo de execução obtidas nos experimentos da solução proposta com as provenientes dos compressores de geometria presentes na literatura. Portanto, a análise dos resultados pretende verificar se o objetivo de desenvolver uma implementação na linguaguem de programação C++ a qual apresente ao menos a mesma taxa de compressão e desempenho superior tratando-se do tempo de execução em relação à implementação original em MATLAB, a fim de tornar o algoritmo mais competitivo, foi cumprido.

Sob essa ótica, foram elencadas comparações entre os resultados do codificador de Octree puro, do TMC13 (MPEG), de versões do Silhouette 3D implementado em MATLAB e da implementação do mesmo algoritmo, descorrida no presente estudo, em C++. É importante evidenciar que a versão elaborada na linguagem de programação C++ possui algumas disparidades técnicas que produzem fluxos de bits diferentes do original em MATLAB. Primeiramente, a inicialização do contexto ocorre de maneira distinta, onde na implementação vigente, todos os contextos presentes na tabela de contextos são inicializados como 1. Uma segunda diferença está atrelada a divergência do processo de arredondamento entre as duas linguagens abordadas. Essa sutil diferença é expressada no fluxo de bits pelo fato de que a codificação aritmética é sensível à precisão da linguagem em detrimento das sucessivas divisões de intervalos.

Dito isso, ainda deve-se levar em consideração que o projeto proposto não desenvolveu o modo SingleMode de codificação para o S3D como descrito em [9], contentando-se apenas na solução diádica do algoritmo. Por outro lado, visto que o modelo com SingleMode é o que mais se aproxima de valores do estado-da-arte em compressão, ainda vamos analisar seus resultados com os do presente estudo.

As medidas relativas ao tempo e taxa de execução do MATLAB e C++ foram realizadas em um computador Desktop com CPU Intel (R) Core (TM) i7-6700K a 4,00 GHz, baseado na arquitetura x64, 32,0 GB de RAM instalada, executando no sistema operacional Windows 10 Education Edition da Microsoft. Os resultados obtidos a seguir são provenientes do processamento das amostras do cojunto de Point Clouds dinâmicas "Microsoft Voxelized Upper Bodies - A Voxelized Point Cloud Dataset", pertencente a base de dados JPEG Pleno fornecida pela Microsoft. [28], avaliadas estáticamente. As Point Clouds avaliadas, Andrew9, David9, Phil9, Ricardo9, Sarah9, são compostas por 9 bits, de dimensão 512 x 512 x 512 cada e representadas, respectivamente, por 318, 216, 245, 216 e 207 quadros. Um quadro de cada sequência de Point Clouds é ilustrado abaixo.

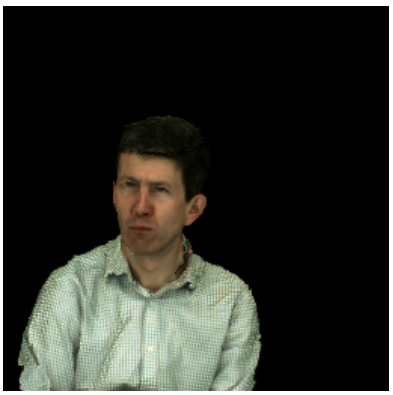

Figura 4.1: Vista renderizada de Point Cloud Andrew9.

Figura 4.2: Vista renderizada de Point Cloud David9.

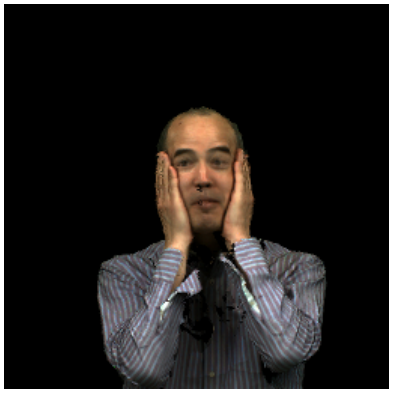

Figura 4.3: Vista renderizada de Point Cloud Phil9.

Figura 4.4: Vista renderizada de Point Cloud Ricardo9.

Figura 4.5: Vista renderizada de Point Cloud Sarah9.

A relação entre os valores de tamanho dos arquivos antes e após a compressão foi omitida em detrimento de que arquivos sem nenhum tipo de compressão serão sempre exageradamente grandes e esse valor não agregaria tanto significado à análise final. Por outro lado, o que é mais interessante de ser notado nos testes é a taxa final de compressão em bits por voxel ocupado (BPVO), pois essa é a taxa que relaciona o espaço de memória ocupado por cada elemento de volume dentro da Point Cloud.

Assim, foram processados os primeiros 20 quadros dentro de cada sequência de Point Cloud apresentada, e através destes, feito a média da taxa de BPVO para cada uma das sequências. Por mais que os

codificadores da literatura expressem a média dos primeiros 200 quadros dentro de cada sequência, ainda é possível comparar os resultados dessas abordagens, com valores fidedignos, conforme ilustrado na Tabela 4.1. É válido ressaltar que os resultados pertinentes aos codificadores que efetuaram a média de 200 quadros foram obtidos diretamente das referências, não foram realizados testes reais para os algoritmos da análise, fora os relacionados ao S3D.

Tabela 4.1: Comparação entre taxas de compressão dos codificadores de geometria.

Sequência	**200 frames (BPVO)**			**S3D C++ 20 frames (BPVO)**		
	Octree	MPEG TMC13	S3D SM	Eixo X	Eixo Y	Eixo Z
Andrew	2.58	1.14	**1.12**	1.23	1.24	**1.21**
David	2.62	1.08	**1.06**	1.23	1.23	**1.23**
Phi	2.64	1.18	**1.14**	**1.25**	1.29	1.26
Ricardo	2.59	1.08	**1.03**	1.11	1.13	**1.11**
Sarah	2.61	1.07	**1.07**	1.22	1.23	**1.20**
Average	2.61	1.11	**1.08**	1.21	1.22	**1.20**

Desta forma, pode-se observar que o S3D implementado na linguaguem de programação C++, mesmo desempenhando de maneira inferior ao codificador de melhor resultado nessa análise, o S3D com SingleMode proposto em 2020 [9], ainda obteve resultados de compressão significativos. Superando o método das Octrees e ficando próximo ao TMC13, devendo na média 8.1% de compressão em relação a este último. Em contrapartida, o fato de que foram processados 10 vezes menos casos de testes, faz com que a média do algoritmo proposto não seja tão precisa quando a dos outros codificadores, tornando possível que essa proximidade fosse ainda mais estreita.

Sob o mesmo conjunto de Point Clouds, ainda foi elaborada a comparação de desempenho entre versões do próprio S3D. Vamos definir S3D Init como o algoritmo original com as tabelas de contextos inicializadas, S3D SM como o algoritmo que implementa o SingleMode e S3D puramente como o algoritmo orinal sem SingleMode ou inicialização da tabela de contextos. Nesse cenário, foi possível verificar que o presente trabalho quando comparado a uma versão equivalente em MATLAB, consegue atuar no mesmo nível de desempenho de compressão conforme apresentado na Tabela 4.2 e superar em nível de tempo de execução conforme apresentado na Tabela 4.3. Em relação ao cálculo do tempo de execução de cada codificador, para cada quadro em cada sequência, foram amostrados 5 tempos. Destes valores, foram descartados o maior e o menor e então feito a média dos 3 que sobraram para finalmente obter o resultado final. Essa estratégia é adotada buscando evadir a questão do sistema operacional poder demorar a estabilizar o acesso a memória durante a codificação. Outra análise interessante que pode ser extraída dessa comparação, é em relação a como o estado em que as tabelas de contexto se encontram no início do algoritmo influenciam a performance deste de maneira considerável. Os resultados do S3D Init tanto em MATLAB quanto em C++ apresentaram uma otimização em relação ao S3D puro onde não houve alteração na inicialização do contexo.

Tabela 4.2: Comparação entre o BPVO médio de versões do algoritmo S3D.

Sequence	**MATLAB (BPVO)**			**C++ (BPVO)**
	S3D	S3D Init	S3D SM	S3D Init
Andrew	1.31	1.23	**1.12**	1.21
David	1.32	1.24	**1.06**	1.22
Phil	1.34	1.27	**1.14**	1.25
Ricardo	1.21	1.12	**1.03**	1.10
Sarah	1.30	1.21	**1.07**	1.20
Average	1.30	1.21	**1.08**	1.20

Tabela 4.3: Comparação entre os tempos dos CODEC da implementação original em MATLAB sem SM e a proposta em C++.

Sequence	**Codificação (seg)**		**Decodificação (seg)**	
	MATLAB	C++	MATLAB	C++
	S3D	S3D Init	S3D	S3D Init
Andrew	28.20	**5.78**	29.28	**3.30**
David	36.53	**6.86**	40.75	**3.36**
Phil	40.91	**7.31**	43.35	**3.57**
Ricardo	24.38	**3.14**	27.07	**2.28**
Sarah	32.18	**6.58**	34.40	**3.51**
Average	32.44	**5.93**	34.97	**3.20**

Tabela 4.4: Comparação percentual entre os ganhos em tempo de execução dos CODEC da implementação original em MATLAB sem SM e a proposta em C++.

Sequence	**Codificação (%)**	**Decodificação (%)**
	C++	C++
Andrew	-79.50	-88.73
David	-81.22	-91.09
Phil	-74.82	-91.76
Ricardo	-87.12	-91.58
Sarah	-79.55	-89.80
Average	-80.44	-90.59

CONCLUSÕES

Após verificar os resultados apresentados, fica evidente que as expectativas de diminuir o tempo de execução do algoritmo atrelado a manter a taxa de compressão foram cumpridas. Obtivemos como resultado uma média de 80.44% a menos de tempo necessário na codificação e de 90.59% a menos na decodificação, mantendo o nível de compressão de 1.21bpvo para o S3D Init e 1.30bpvo para o S3D implementados em MATLAB contra 1.20bpvo do mesmo algoritmo implementado na linguagem de programação C++.

5 CONCLUSÃO

O Silhouette 3D revelou-se um algoritmo promissor, o qual divide uma Point Cloud em intervalos diádicos a fim de montar uma árvore de silhuetas 2D e através de operações lógicas, deduzir quais símbolos devem ser descartados e quais devem ser enviados ao codificador aritmético. Este, por sua vez, auxiliado por contextos provenientes da árvore de silhuetas, aumenta ainda mais a taxa de compressão do algoritmo. Assim após a compreensão do processo descrito, a análise das estatísticas relacionadas a esse algoritmo permitiu a identificação dos défices de execução do mesmo devido ao ambiente de implementação escolhido. Sob esse viés, a proposta de migração foi implementada e pode-se concretizar os objetivos levantados durante sua elaboração. Ambos os resultados, taxa de compressão equivalente e desempenho de tempo de execução superior, descritos no Capítulo 4, verificam os objetivos do projeto, indicando a realização na criação de uma versão com melhor desempenho do algoritmo.

5.1 TRABALHOS FUTUROS

A partir dessas análises, alguns problemas observados merecem um estudo mais aprofundado e testes em trabalhos futuros. Essas questões dizem respeito a possíveis novos recursos e abordagens do algoritmo, bem como ao trabalho de otimização. A implementação do SingleMode no projeto desenvolvido na linguagem de programação C++, pode alavancar o desempenho do algoritmo proposto tornando-o ainda mais competitivo. As análises entre as versões de MATLAB e C++ nos levam para esse desfecho, porém ainda não é suficiente para inferir que o SingleMode pode de fato otimizar o algoritmo de todas as maneiras desejadas. Além disso, releituras de funções implementadas que são custosas para o algoritmo a fim de refiná-las e reduzir seu custo, também são interessante em uma análise mais aprofundada deste trabalho. Sob outra perspectiva, revela-se factível a ideia de realizar codificações parciais na estrutura da Point Cloud, permitindo que essa seja comprimida paralalemente em prol de unidades de contexto permitiram que a Point Cloud fosse comprimida ao vivo, por pacotes, durante transmissões.

REFERÊNCIAS BIBLIOGRÁFICAS

1 QUEIROZ, R. L. de; CHOU, P. A. Compression of 3d point clouds using a region-adaptive hierarchical transform. *IEEE Transactions on Image Processing*, v. 25, n. 8, p. 3947–3956, 2016.

2 HOANG, L.; LEE, S.-H.; KWON, O.-H.; KWON, K.-R. A deep learning method for 3d object classification using the wave kernel signature and a center point of the 3d-triangle mesh. *Electronics*, v. 8, n. 10, 2019. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/8/10/1196>.

3 XU, Y.; TONG, X.; STILLA, U. Voxel-based representation of 3d point clouds: Methods, applications, and its potential use in the construction industry. *Automation in Construction*, v. 126, p. 103675, 2021. ISSN 0926-5805. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0926580521001266>.

4 PENG, J.; KUO, C. C. J. Octree-based progressive geometry encoder. In: SMITH, J. R.; PANCHANATHAN, S.; ZHANG, T. (Ed.). *Internet Multimedia Management Systems IV*. SPIE, 2003. v. 5242, p. 301 – 311. Disponível em: <https://doi.org/10.1117/12.510857>.

5 TATARCHENKO, M.; DOSOVITSKIY, A.; BROX, T. Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. *2017 IEEE International Conference on Computer Vision (ICCV)*, p. 2107–2115, 2017.

6 DISTRIBUIçõES de Probabilidade Discreta. Disponível em: <https://biostatistics-uem.github.io/Bio/probabilidade.html>.

7 SIM, J.-Y.; LEE, S.-U. Compression of 3-d point visual data using vector quantization and rate-distortion optimization. *IEEE Transactions on Multimedia*, v. 10, n. 3, p. 305–315, 2008.

8 MPEG Home Page. <https://www.mpegstandards.org/>. Accessed: 2021-11-08.

9 PEIXOTO, E. Intra-frame compression of point cloud geometry using dyadic decomposition. *IEEE Signal Processing Letters*, v. 27, p. 246–250, 2020.

10 YANG, B.; XU, W.; DONG, Z. Automated extraction of building outlines from airborne laser scanning point clouds. *IEEE Geoscience and Remote Sensing Letters*, v. 10, n. 6, p. 1399–1403, 2013.

11 HUANG, R.; HONG, D.; XU, Y.; YAO, W.; STILLA, U. Multi-scale local context embedding for lidar point cloud classification. *IEEE Geoscience and Remote Sensing Letters*, v. 17, n. 4, p. 721–725, 2020.

12 TANG, P.; HUBER, D.; AKINCI, B.; LIPMAN, R.; LYTLE, A. Automatic reconstruction of as-built building information models from laser-scanned point clouds: A review of related techniques. *Automation in Construction*, v. 19, n. 7, p. 829–843, 2010. ISSN 0926-5805. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0926580510000907>.

13 HAN, J.; SHAO, L.; XU, D.; SHOTTON, J. Enhanced computer vision with microsoft kinect sensor: A review. *IEEE Transactions on Cybernetics*, v. 43, n. 5, p. 1318–1334, 2013.

14 PARK, J.; KIM, H.; TAI, Y.-W.; BROWN, M. S.; KWEON, I. High quality depth map upsampling for 3d-tof cameras. In: *2011 International Conference on Computer Vision*. [S.l.: s.n.], 2011. p. 1623–1630.

15 HAN, X.-F.; JIN, J. S.; WANG, M.-J.; JIANG, W.; GAO, L.; XIAO, L. A review of algorithms for filtering the 3d point cloud. *Signal Processing: Image Communication*, v. 57, p. 103–112, 2017. ISSN 0923-5965. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0923596517300930>.

16 SAYOOD, K. *Introduction to Data Compression*. Department of Electrical Engineering, University Nebraska-Lincoln, Lincoln, Nebraska: ACADEMIC PRESS, 2003.

17 SCHNABEL, R.; KLEIN, R. Octree-based point-cloud compression. In: *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*. Goslar, DEU: Eurographics Association, 2006. (SPBG'06), p. 111–121. ISBN 3905673320.

18 TANG, L.; DA, F. peng; HUANG, Y. Compression algorithm of scattered point cloud based on octree coding. In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. [S.l.: s.n.], 2016. p. 85–89.

19 ELLIS, R. B.; KAHNG, A. B.; ZHENG, Y. Compression algorithms for dummy-fill VLSI layout data. In: STARIKOV, A. (Ed.). *Design and Process Integration for Microelectronic Manufacturing*. SPIE, 2003. v. 5042, p. 233 – 245. Disponível em: <https://doi.org/10.1117/12.485247>.

20 TILLEY, S.; SMITH, D. *Perspectives on Legacy System Reengineering*. 1995.

21 KAZMAN, R.; WOODS, S.; CARRIERE, S. Requirements for integrating software architecture and reengineering models: Corum ii. In: *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*. [S.l.: s.n.], 1998. p. 154–163.

22 A successful Git branching model by Vincent Driessen. 2010. <https://nvie.com/posts/a-successful-git-branching-model/>. Accessed: 2021-11-09.

23 GOOGLE C++ Style Guide. <https://google.github.io/styleguide/cppguide.html>. Accessed: 2021-09-23.

24 OSHEROVE, R. *The Art of Unit Testing*. [S.l.]: Manning Publications, 2013.

25 GAROUSI, V.; MäNTYLä, M. V. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, v. 76, p. 92–117, 2016. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584916300702>.

26 DOXYGEN. <https://www.doxygen.nl/manual/docblocks.html>. Accessed: 2021-11-09.

27 UNB: Geometry Coder Project. <https://github.com/pointcloud-unb/geometryCoderCpp>. Accessed: 2021-11-18.

28 JPEG Pleno Database: Microsoft Voxelized Upper Bodies - A Voxelized Point Cloud Dataset. <http://plenodb.jpeg.org/pc/microsoft>. Accessed: 2021-08-29.