UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

JOÃO VITOR BREDA

DESENVOLVIMENTO DE TERRENO PROCEDURAL INFINITO EM VOXEL UTILIZANDO C++ E OPENGL

DOIS VIZINHOS

2022

JOÃO VITOR BREDA

DESENVOLVIMENTO DE TERRENO PROCEDURAL INFINITO EM VOXEL UTILIZANDO C++ E OPENGL

DEVELOPMENT OF INFINITE PROCEDURAL TERRAIN IN VOXEL USING C++ AND OPENGL

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Orientador: Profa. Dra. Simone de Sousa Borges

DOIS VIZINHOS
2022

JOÃO VITOR BREDA

DESENVOLVIMENTO DE TERRENO PROCEDURAL INFINITO EM VOXEL UTILIZANDO C++ E OPENGL

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Software do Curso de Bacharelado em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 05/dezembro/2022

Simone de Sousa Borges
doutorado
Universidade Tecnológica Federal do Paraná

Marco Antônio Simões Teixeira
doutorado
Universidade Tecnológica Federal do Paraná

Marisângela Pacheco Brittes
doutorado
Universidade Tecnológica Federal do Paraná

DOIS VIZINHOS
2022

AGRADECIMENTOS

A todos que de alguma forma auxiliaram em minha formação, em especial à Prof.ª Dr.ª Simone de Sousa Borges por todo o apoio prestado para a realização deste trabalho, o meu muito obrigado.

RESUMO

Algoritmos de geração procedural com voxels são utilizados para gerar dados e com base neles criar terrenos em um ambiente virtual em três dimensões. Além disso, esses terrenos podem aparentar serem infinitos pelo seu enorme tamanho. Este tipo de aplicação carece de documentos e exemplos detalhando e disponibilizando sua utilização, usualmente sendo implementados no formato caixa-preta nas ferramentas de modelagem disponíveis. Visando auxiliar a utilização desta tecnologia, este trabalho apresenta o resultado de uma pesquisa para o desenvolvimento de uma aplicação que faça a geração de terreno procedural infinito em voxel, utilizando a melhor abordagem encontrada no estado da arte atual e documentando-a. Contribuindo assim, para que outros interessados possam utilizar o protótipo desenvolvido ou se basear no conteúdo deste trabalho para auxiliar no desenvolvimento de suas próprias aplicações para representar volumes com voxels.

Palavras-chave: voxel; geração procedural; geração de terreno; transferência de tecnologia; engenharia de software.

ABSTRACT

Procedural generation algorithms with voxels are used to generate data and based on them create terrains in a virtual three-dimensional environment. In addition, these terrains can appear to be infinite due to their enormous size. This type of application lacks documents and examples detailing and providing its use, usually being implemented with black box format in the available modeling tools. Aiming to help the use of this technology, this work presents the result of a research for the development of an application that generates an infinite procedural terrain in voxel, using the best approach found in the current state of the art and documenting it. Thus, contributing so that other interested parties can use the developed prototype or base themselves on the content of this work to assist in the development of their own applications to represent volume with voxels.

Keywords: voxel; procedural generation; terrain generation; technology transfer; software engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGEM DE CÓDIGOS FONTE

SUMÁRIO

1 INTRODUÇÃO

Este Capítulo visa apresentar de forma sucinta, os elementos que compõem este trabalho, como a contextualização, abordada na Seção 1.1, motivação na Seção **??**, objetivos, sendo abordados na Seção 1.2 e por último, a estrutura deste trabalho é descrito na Seção **??**.

1.1 Contextualização

Voxels são cubóides[1] utilizados para representar volume em um ambiente de 3 dimensões, podendo ser aplicados para diversos fins, como visualização de gráficos, imagens médicas e jogos digitais (CHEN; KAUFMAN ARIE E. ANDYAGEL, 2000). Embora possuam diversos fins, sua utilização foi limitada durante muito tempo devido à grande quantidade de memória que é requerida ao utilizá-los, porém, como a quantidade de processamento e memória computacional aumentou imensamente nos últimos anos, os voxels estão se tornando uma aplicação popular novamente (DUSTERWALD, 2015).

Esta popularidade vem se mostrando na utilização de voxels em filmes e televisão, para efeitos volumétricos como de nuvens e simulação fluída, sendo aplicados em grandes filmes como O Senhor dos Anéis e O Dia Depois de Amanhã (DUSTERWALD, 2015). Além disso, popularizado pelo jogo de sucesso Minecraft, voxels se tornaram um gênero de jogos digitais que tem como apelo a possibilidade de completa manipulação do mundo pelos jogadores (DUSTERWALD, 2015).

Voxels podem ser utilizados em conjunto com geração procedural, um tipo de algoritmo que gera dados. Utilizando esses dados para definir onde gerar voxels (podendo eles serem de diversos tipos), é possível criar terrenos de forma procedural, ou seja, não há necessidade de definir onde gerar ou não gerar voxels, sendo isso definido pelos dados gerados pelo algoritmo procedural, o que possibilita a criação de terrenos “aparentemente infinitos”, mas que na realidade são finitos de acordo com o espaço de memória disponível, mas que podem ser tão grandes que aparentam serem infinitos aos usuários (BRUMMELEN; CHEN, 2018).

Estes terrenos “infinitos” em voxels são principalmente valiosos para a aplicação em jogos digitais, pois simplificam o trabalho dos desenvolvedores na criação de terreno, reduzindo custos (SHORT; ADAMS, 2017). Porém, outras aplicações podem se utilizar desta forma de gerar uma maior quantidade de voxels, apenas utilizando dados próprios ao invés dos gerados proceduralmente.

[1] Cubóide - Poliedro com seis faces retangulares.

1.2 Objetivo Geral

O objetivo geral é desenvolver um sistema para dispositivos móveis multiplataforma, fundamentado no uso da geolocalização, para possibilitar a comunicação e acompanhamento de entregas com transportadores independentes.

1.3 Objetivos Específicos

Baseando-se no objetivo geral foram definidos os seguintes objetivos específicos:

1. Estudar e realizar o levantamento dos requisitos;
2. Analisar as características e capacidades do framework Flutter;
3. Integrar a API de Geolocalização;
4. Criar um protótipo inicial no produto;
5. Desenvolver e testar a solução proposta;

1.4 Estrutura da Monografia

Este trabalho está organizado em sete capítulos principais da seguinte forma: O capítulo 2 apresenta conceitos e definições acerca dos temas que serão utilizados neste trabalho. No Capítulo 3 é apresentado o mapeamento sistemático realizado para encontrar os principais estudos em torno do tema e encontrar a melhor abordagem a ser utilizada. No Capítulo 4 é mostrada a proposta deste trabalho. A metodologia é definida no capítulo 5onde são detalhadas as técnicas identificadas. O Capítulo 6 apresenta a documentação do desenvolvimento do algoritmo proposto. O Capítulo 7 demonstra os resultados atingidos neste trabalho. Por fim, o Capítulo 8 apresenta as considerações finais.

2 ASPECTOS CONCEITUAS

Este Capítulo visa apresentar os conceitos necessários para o entendimento do trabalho proposto e também sua contextualização com a revisão da literatura.

2.0.1 Computação Gráfica

A computação gráfica é uma subárea da ciência da computação, mas que envolve várias outras áreas das mais diversas disciplinas, como arte, ciência, produção de filmes, dentre outras, ou seja, sua utilização não é específica à indústria da computação, mas é sim utilizada amplamente em várias indústrias, como por exemplo educação, entretenimento e negócios. Alguns dos tipos de aplicações mais específicas são (FOLEY, 1993):

- Interfaces gráficas
- Cartografia, para representação de dados geográficos
- Medicina, para gerar imagens que ajudam os médicos a guiar os instrumentos
- Design de componentes, de diversas áreas como construção civil e indústria automobilística
- Animação de filmes

2.0.1.1 Renderização

Renderização é o processo que utiliza as informações de uma cena, como polígonos e iluminação, e calcula o resultado final para gerar uma imagem formada por pixels. O processo de renderização segue uma *pipeline* gráfica que pode diferenciar dependendo das ferramentas a serem utilizadas (CHEN; KAUFMAN ARIE E. ANDYAGEL, 2000).

2.0.1.2 Rasterização

Rasterização é uma importante parte do processo de renderização que envolve a transformação de um desenho vetorial (uma imagem com curvas) em um array 2D de pixels (CHEN; KAUFMAN ARIE E. ANDYAGEL, 2000).

2.0.1.3 Malhas

Uma malha, ou malha poligonal, é um conjunto de vértices (ponto em que duas ou mais retas se encontram) que definem o formato de um objeto, similar ao processo de tesselação.

Malhas são comumente utilizadas na computação gráfica em razão de sua modelagem e renderização eficiente de objetos 3D (CBOVIK, 2005).

2.0.2 OPENGL

De acordo com o site oficial mantido pelo Khronos (2022), o OpenGL é uma API para renderização de gráficos e interação com a GPU que não requer instalação do usuário, pois de modo simples, se trata de uma especificação mantida pelo Khronos Group e implementada pelas fabricantes de placas de vídeo. Desse modo, cada placa de vídeo pode suportar diferentes versões do OpenGL, estando a requisito do fabricante o lançamento de drivers para suportar novas versões.

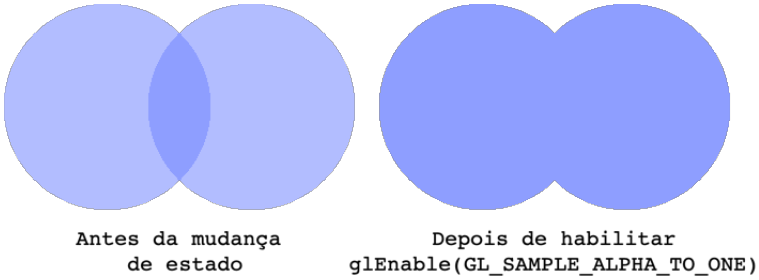
Até sua versão 3.2, o OpenGL constava apenas com um modo de desenvolvimento, chamado de *immediate mode*. Neste modo, a renderização de gráficos era realizada de uma forma simplificada, porém em contraste, os desenvolvedores não tinham muito controle sobre várias funcionalidades do OpenGL que definem como ele deve operar (VRIES, 2020).

A partir de sua versão 3.2, o *immediate mode* foi descontinuado a favor de um novo modo chamado *core-profile mode*. Neste modo, o OpenGL força o desenvolvedor a usar práticas modernas, sendo mais flexível e eficiente, ao custo de sua utilização ser mais difícil (VRIES, 2020).

Ainda de acordo com Vries (2020), para desenvolver com o OpenGL é importante notar que seu funcionamento é considerado uma máquina de estados, ou seja, uma coleção de variáveis que definem como o OpenGL deve operar. Seu funcionamento básico envolve a mudança de suas variáveis de estado e então a utilização de funções que utilizem esse estado.

A figura 1 a seguir demonstra um exemplo de mudança de estado, onde é utilizada uma função do OpenGL para mudar um de seus atributos de estado, que faz com que a renderização utilize o valor máximo de opacidade (KHRONOS, 2022). Depois de mudar esse atributo de estado, todas as renderizações seguintes são feitas dessa forma até que esse estado seja mudado.

Figura 1 – Exemplo de mudança de estado no OpenGL

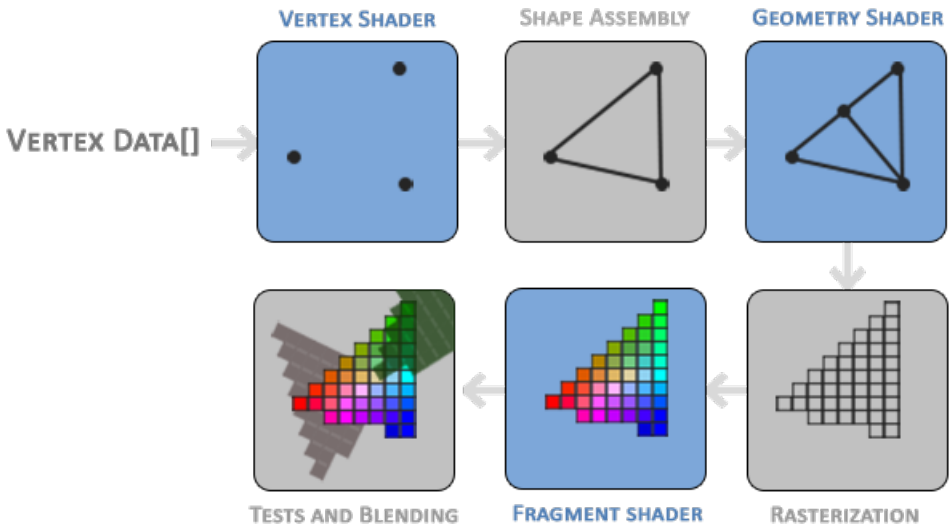


Fonte: Autoria própria.

2.0.2.1 Pipeline Gráfica do OpenGL 3.3 e Shaders

Como mencionado na Seção 2.1.1, a *pipeline* gráfica define as etapas do processo de renderização. No OpenGL, tudo é considerado como estando em um ambiente 3D, mas para representar essa informação na tela é necessário sua conversão para um *array* 2D de pixels (VRIES, 2020). Dentro do processo da *pipeline* gráfica, o usuário pode projetar programas, chamados de *shaders*, que fazem a computação de dados na GPU (KHRONOS, 2022). A Figura 2 a seguir demonstra a pipeline gráfica do OpenGL 3.3, onde as etapas que podem ser modificadas pelos desenvolvedores estão marcadas com fundo em azul (VRIES, 2020).

Figura 2 – Pipeline gráfica do OpenGL 3.3



Fonte: Vries (2020).

- *Vertex Shader:* Programa que recebe como entrada um vértice, ou seja, uma coleção de dados por coordenada 3D, e realiza um processamento definido pelo usuário no mesmo, como por exemplo uma alteração de cores.
- *Shape Assembly (ou Primitive Assembly):* Esse estágio tem como entrada todos os vértices recebidos do vertex shader que formam uma primitiva gráfica (elementos básicos como linhas e polígonos), juntando todos os pontos para formar a primitiva definida (no caso da Figura 1, um triângulo).
- *Geometry Shader:* Programa que recebe como entrada uma coleção de vértices que formam uma primitiva recebida do *shape assembly*, e então realiza um pós-processamento dos vértices, podendo por exemplo gerar uma nova primitiva como na Figura 1. É a etapa que governa o processamento de primitivas.
- *Rasterization:* Conforme já explicado na Seção 2.1.2, é onde é realizada a conversão das primitivas para os pixels equivalentes na tela, resultando em fragmentos. É impor-

tante notar que nesta etapa todos os fragmentos que estão fora de visão são descartados para aumentar a performance, num processo conhecido como *clipping*.

- *Fragment Shader:* Etapa que recebe como entrada os fragmentos resultantes da rasterização e calcula cor final dos pixels, além de quaisquer outros processamentos definidos pelo usuário.
- *Tests and Blending:* Etapa final que faz a interpolação dos pixels na tela de acordo com profundidade e opacidade.

2.0.2.2 VBOs, VAOs e EBOs

VBOs, VAOs e EBOs são componentes do OpenGL para manipulação de memória. Mais especificamente, o *Vertex Buffer Object* (VBO) possibilita o usuário de enviar grandes quantidades de dados (vértices) para a GPU de uma só vez, podendo armazená-los na memória da GPU. Depois de armazenados, os dados podem ser acessados pelo vertex shader de forma muito rápida (VRIES, 2020).

Já o *Vertex Array Object* (VAO) é utilizado para armazenar os dados de atributos de vértices, tendo como vantagem que o processo de configuração de atributos só precisa ser realizado uma vez, após isso basta vincular o VAO correspondente ao OpenGL (VRIES, 2020).

Por fim, o *Element Buffer Object* (EBO) realiza o armazenamento de índices que definem quais vértices devem ser renderizados, sendo utilizado em conjunto com VBOs que armazenam apenas vértices únicos. Sua vantagem é não necessitar o armazenamento de vértices que se sobrepõem, ou seja, que são iguais, mas que são utilizados para formar diferentes primitivas. Estes índices definem quais vértices devem ser utilizados para formar cada primitiva (VRIES, 2020).

2.0.3 Geração Procedural

Voxels podem ser utilizados em conjunto com geração procedural, um tipo de algoritmo que gera dados. Principalmente quando aplicado a jogos, a geração de terreno com voxels é feita de forma procedural, o que simplifica o trabalho dos desenvolvedores ao não precisar criar o terreno manualmente e possibilita a oportunidade de criar terrenos “aparentemente infinitos”, ou seja, que podem ser tão grandes que aparentam serem infinitos aos usuários, mas que na realidade são finitos de acordo com o espaço de memória disponível (BRUMMELEN; CHEN, 2018).

2.0.4 Voxel

Aplicações baseadas em gráficos utilizam imagens rasterizadas para exibir componentes, sendo formadas por bitmaps, que são *arrays* binários retangulares de pontos chamados de pixels, oriundo das palavras *picture elements* (FOLEY, 1993). Pelo fato da maioria dos monitores de computadores serem aptos a demonstrar apenas imagens de bitmap 2D, todas as imagens, mesmo as aparentemente 3D, são rasterizadas na maioria das vezes como triângulos e projetadas em espaço 2D (WILDER, 2015).

Quando falamos de imagens 2D, nós a definimos como um conjunto de pixels, ou *picture elements*, mas se elevarmos a dimensão para 3D então podemos formar uma imagem com voxels, oriundo das palavras *volume elements*, que são especificamente cubóides retangulares com seis faces, doze arestas e oito cantos (CHEN; KAUFMAN ARIE E. ANDYAGEL, 2000). Utilizando voxels, é possível representar volume para diversos fins, como visualização de gráficos, imagens médicas e mais recentemente popularizado, jogos digitais (CHEN; KAUFMAN ARIE E. ANDYAGEL, 2000).

3 MAPEAMENTO SISTEMÁTICO

Este Capítulo visa apresentar os estudos relacionados com desenvolvimento de terreno procedural infinito em voxel. Na Seção 3.1 é descrito o processo sistemático utilizado para conduzir o mapeamento. Já na Seção 3.2 são analisados os resultados obtidos no processo anterior. Por fim, é apresentado as ameaças à validade do mapeamento na Seção 3.3 e as considerações finais na Seção 3.4.

3.0.1 Proceso do Mapeamento

O mapeamento sistemático foi conduzido conforme o processo descrito por Petersen, Vakkalanka e Kuzniarz (2015), do qual organiza todo o desenvolvimento em cinco passos principais, sendo eles: (i) definição de questões de pesquisa, (ii) realização da pesquisa de estudos primários relevantes, (iii) triagem dos documentos, (iv) *keywording* dos resumos, e (v) a extração de dados e mapeamento.

Conforme o primeiro passo, às seguintes questões de pesquisa foram definidas para exemplificar os objetivos do estudo do mapeamento:

QP1: Como é realizada a criação de terrenos procedurais infinitos em voxels?

QP2: Quais técnicas são utilizadas na criação de terrenos em voxels?

QP3: Quais tecnologias (linguagem de programação, biblioteca gráfica) são mais utilizadas na criação de terrenos em voxels e quais são seus prós e contras?

Para realizar a busca de forma satisfatória, foram realizados alguns testes para identificar quais palavras-chave seriam utilizadas na *string* de busca. Inicialmente, os testes consistiram na combinação das palavras "voxel", "terreno" e "geração procedural", todas no idioma inglês. Eventualmente, a palavra "*game*" também foi incluída para englobar artigos que descrevem o processo de utilização de voxels em motores de jogos, conteúdo o qual seria de grande ajuda para esta pesquisa.

Devido à grande volatilidade no número de artigos retornados conforme a *string* era formulada, com alguns motores retornando muitos artigos e outros pouquíssimos, a *string* de busca foi definida utilizando as principais palavras-chave para esta pesquisa, juntamente a aplicação de filtros nos motores conforme a necessidade para que fosse realizada uma segunda filtragem. A *string* de busca utilizada nesta etapa foi:

(voxel) AND (game OR engine OR procedural generation OR terrain)

As buscas então foram realizadas nas bases de dados eletrônicas apresentadas na Tabela 1, realizadas no mês de abril de 2021 conforme o seguinte:

Na IEEE, a *string* foi aplicada normalmente e sem filtros; Na ACM, a *string* foi aplicada apenas no *abstract*, porém, mudando a fonte dos resultados de "*ACM Full-Text Collection*" para "*ACM Guide to Computing Literature*", da qual engloba mais conteúdo de computação; Na Science Direct, a *string*foi aplicada normalmente, porém, utilizando o filtro "*Publication title*" para filtrar apenas "*Computer & Graphics*", assim evitando a maioria dos artigos de âmbito médico que retornavam; Por fim na Springer, a *string*foi aplicada normalmente, porém, utilizando os seguintes filtros: "*Content Type: Article*", "*Discipline: Computer Science*", "*Subdiscipline: Computer Graphics*" e "*Language: English*".

Tabela 1 – Resultados retornados por base de dados

Base de dados	Quantidade
ACM Digital Library	123
Elsevier (Science Direct)	135
IEEE Xplore	116
Springer	204
Total	578
Candidatos	52
Seleção Final	3

Fonte: Autoria própria.

Conforme o processo descrito anteriormente, foram realizadas duas triagens dos estudos primários retornados, visando identificar os relevantes para responder às questões de pesquisa.

A primeira triagem foi realizada com a leitura dos títulos, resumos e palavras-chave, reduzindo o conjunto inicial dos estudos de 578 para 52 candidatos. Durante essa triagem, foram aplicados critérios de exclusão, sendo eles:

- Artigo não explica como fazer a implementação de um algoritmo utilizando voxels.
- Artigo que tenha como foco o uso de voxels para aplicações específicas que não auxiliam no uso geral de voxels.

Na segunda triagem fora realizada a leitura dos resumos, introdução e conclusão além de uma leitura estilo *skimming* nos demais conteúdos, novamente aplicando os critérios de exclusão nos 52 candidatos, tendo como resultado um subconjunto de 3 estudos primários.

É importante ressaltar que o motivo da seleção final conter poucos resultados provêm do fato de que a utilização da palavra-chave "voxel", muito importante para esta pesquisa, possui conotação diferente em outras áreas de pesquisa, o que acabou por retornar vários estudos relacionados ao processo de "voxelização", o qual se refere ao processo de conversão de objetos geométricos de sua representação geométrica contínua (como uma malha triangular 3D), em um conjunto de voxels (um grid 2D) que se aproxima a isso, conforme escrito por Ogayar, Rueda e Segura (2007).

Considerando este fato de que o processo de "voxelização" tem como foco a conversão de dados para dados representáveis por voxels, esta pesquisa acaba por não se beneficiar desse

conhecimento, pois se atém mais especificamente sobre a utilização dos voxels em um ambiente 3D, e por essa razão, estes artigos foram eliminados.

Por fim, os 3 estudos que compõem a seleção final foram lidos na íntegra. Devido a baixa quantidade de estudos finais, o processo de *keywording* para a classificação dos estudos não se mostrou necessário.

3.0.2 Análise

O objetivo desta seção é apresentar os detalhes dos estudos selecionados, de forma clara e sucinta, a fim de responder às questões de pesquisa da forma mais satisfatória possível.

Em razão do pequeno número de estudos selecionados, estudos adicionais também tiveram algumas informações resumidas, provenientes de estudos encontrados previamente ao processo de mapeamento no conjunto inicial de pesquisa que incentivou a criação deste trabalho. Estes estudos foram encontrados em páginas de universidades e afins, não tendo um nível de qualidade garantido por revistas de publicação, e por esse ou outros motivos acabaram por não estarem no conjunto de estudos nas bases de dados selecionadas.

3.0.2.1 Síntese de Estudos do Processo

No primeiro artigo, "A Relevant Research on the Establishment of a Voxel Gaming World" (GAO; HE; QI, 2018), os autores comentam os benefícios de se utilizar voxels para a representação dos mundos em jogos eletrônicos, dentre os quais podemos citar principalmente:

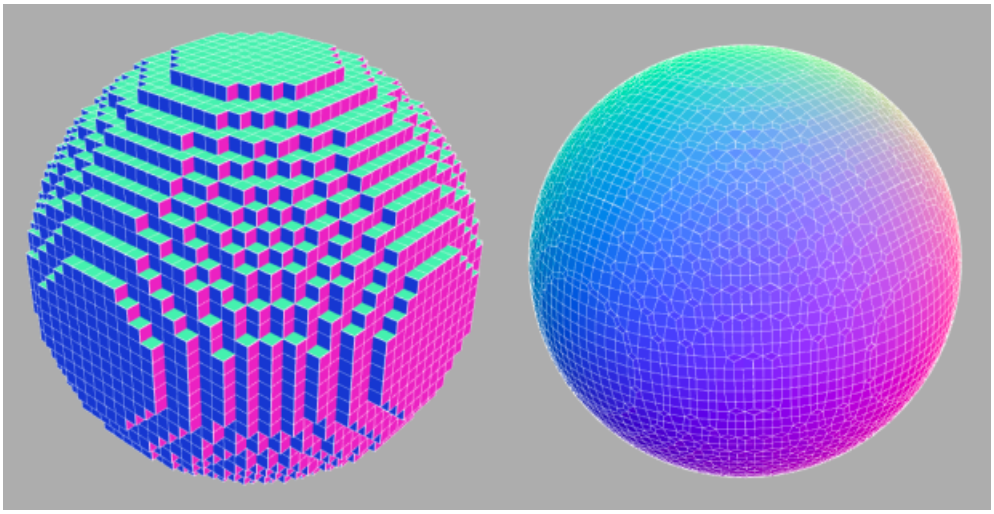
- Custos: A geração de terrenos geralmente corresponde a uma grande quantidade dos custos totais para se produzir um jogo eletrônico. Utilizando voxels e geração procedural, estes custos podem ser evitados pois os terrenos não precisam ser criados inteiramente por pessoas.
- Interatividade: As formas de representação de terrenos tradicionais em jogos, utilizando malhas triangulares em 3D, são difíceis de se modificar ativamente ao mesmo tempo em que mantêm seu nível de detalhe. Já os terrenos compostos por voxels possibilitam sua manipulação com facilidade, pois a forma geométrica simples dos voxels simplifica a alteração das malhas dos terrenos.

Um fato importante da qual o artigo trata é a de que existem vários tutoriais na internet para a utilização de voxels, porém, os desafios principais envolvem uma renderização eficiente. Além disso, uma grande consideração a ser feita é sobre as tecnologias a serem utilizadas, pois por exemplo, o artigo cita OpenGL como uma tecnologia que pode facilitar a integração de técnicas como simulação de fluídos.

O segundo artigo, "A Real-Time Sculpting and Terrain Generation System for Interactive Content Creation" (CHIEN-WEN *et al.*, 2021) fala sobre a criação de um sistema de modelagem utilizando realidade virtual, ou seja, um sistema que possibilita ao usuário editar conteúdo 3D em um ambiente 3D.

Embora o enfoque do artigo não tenha muita relação e tenha poucos detalhes de implementação, ele trata sobre o uso de voxels, especificamente utilizando a técnica conhecida como "*smooth* voxels", na qual a malha dos voxels é interconectada para criar uma forma mais suave (3).

Figura 3 – Imagem de uma esfera em voxels (esquerda) e a mesma após aplicado o processo de smooth voxels (direita)



Fonte: LYSENKO (2012c).

O conteúdo mais útil para esta pesquisa encontrado neste artigo é a definição de sua pipeline de geração de volume, citando principalmente a utilização de *chunks*, onde cada *chunk* é basicamente um pedaço de todo o volume.

A separação do terreno em *chunks* é de suma importância na geração de terrenos com voxels, pois em um terreno interativo, uma pequena modificação nos voxels acarretaria apenas na reconstrução da malha deste *chunk* e o reenvio somente desses dados para a renderização. Sem a separação do terreno em *chunks*, podemos considerar dois cenários:

- A utilização de uma malha para todo o terreno, que nos casos onde o terreno tem um tamanho consideravelmente grande, como os terrenos pseudo-infinitos abordados nesta pesquisa, uma pequena modificação resultaria na reconstrução da malha de todo o terreno e no reenvio de todos os dados para a renderização, o que seria muito custoso e possivelmente inviabilizaria sua utilização.

- A utilização de uma malha para cada voxel, onde a reconstrução de sua malha não seria um problema, mas as várias requisições dos dados de cada um dos voxels para renderização também seria muito custosa e possivelmente inviabilizaria sua utilização.

Esse processo de aglomerar dados para enviá-los em conjunto e assim ter menos requisições de renderização é conhecido como *geometry batching*, sendo muito utilizado pois possibilita à GPU atingir uma performance muito melhor, conforme dito por TRAPP e DÖLLNER (2015).

E finalmente, no terceiro e último artigo selecionado, "Procedural Approach to Volumetric Terrain Generation" (SANTAMARÍA-IBIRIKA *et al.*, 2014), os autores tem como foco a geração procedural de terrenos com voxels, especialmente de sua parte subterrânea, procurando realizar a geração de minérios e afins nesses terrenos de forma realista.

Embora o artigo não tenha detalhes de implementação da geração de terrenos em voxels, seu foco teórico em como realizar uma geração procedural desses terrenos, considerando ainda a utilização de *chunks*, de forma mais realista pode ser útil para algumas implementações.

O processo de geração procedural descrito no artigo tem como principal funcionamento a definição de vários parâmetros antes de sua utilização, para que então a criação de camadas, veias de minérios e cavernas seja realizada, de baixo para cima, uma camada de cada vez, onde as chances do que irá gerar são determinadas pelos parâmetros antes definidos.

3.0.2.2 Síntese e Estudos do Conjunto Inicial

Como descrito anteriormente, alguns estudos foram encontrados previamente ao processo de mapeamento. Esta subseção visa sintetizar informações importantes desses estudos para auxílio nesta pesquisa. É importante notar que todos esses estudos têm como foco a geração de terrenos procedurais feitos com voxels.

No projeto final de bacharelado “Game Engine with 3D Graphics” (OLIVEIRA; OLIVEIRA; PEDRINI, 2016), os autores optaram por utilizar C++ e OpenGL devido à sua popularidade na utilização de motores de jogos, e por consequência, maior quantidade de recursos disponíveis. O projeto detalha os aspectos conceituais necessários para sua compreensão, bem como a estruturação do projeto, porém, não há detalhes de implementação nem a disponibilização de um código fonte.

No artigo “An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain” (WILDER, 2015), o autor explica sua decisão em utilizar C++ por ser uma linguagem de baixo nível suficiente para ter manipulação direta de recursos como memória, mas de nível alto o suficiente para possibilitar a utilização de recursos como abstração, polimorfismo e outros. Já sua decisão em utilizar OpenGL é em razão de ser multi-plataforma, robusta, eficiente e gratuita. Em seu artigo ainda, alguns detalhes de implementação são demonstrados, porém, não há código fonte disponível. Além disso, o método aplicado possui limitações comentadas pelo próprio autor, que sugere modificações para trabalhos futuros. Dentro dessas limitações, a mais importante é a de que o método utilizado carrega todo o terreno na memória ao mesmo tempo, ou seja, apenas terreno que caiba na memória RAM pode ser renderizado.

Na tese de bacharelado “Bloxel - Developing a voxel game engine in Java using OpenGL” (**??**), por preferência pessoal dos autores a aplicação é desenvolvida na linguagem de programação Java e utilizando OpenGL. Nesta tese, os autores detalham em teoria várias partes avançadas que um motor de jogos possui, como manipulação de áudio, inteligência artificial, dentre outros. Infelizmente, o projeto se atém à teoria e não detalha o desenvolvimento do código em suas partes e nem tem a disponibilização do código fonte.

Na tese de bacharelado “Voxel Game Engine Using Blender Game Engine” (KIRSI, 2016), embora o código fonte esteja disponível, o autor optou por utilizar um motor de jogos para auxiliar no desenvolvimento da aplicação, o que delimita o projeto à utilização de um software proprietário específico. Em sua tese, porém, ele aborda algumas questões importantes como o que utilizar para armazenar e gerenciar os voxels na memória, optando pela utilização de *hashmaps* contra outras opções como *octrees* e *arrays*. Além disso, é discutida técnicas importantes na hora de gerar a malha do terreno como *culling* e *greedy meshing*.

3.0.3 Ameaças à Validade

Para garantir um processo de seleção de artigos de forma imparcial, as questões de pesquisa e os critérios de inclusão e exclusão foram definidos no início do mapeamento. Além disso, houve um número limitado de base de dados escolhida, assim, considera-se a possibilidade de haver estudos relevantes não incluídos no mapeamento, disponíveis em bases de dados não utilizadas neste estudo.

3.0.4 Considerações Finais

O principal objetivo deste mapeamento sistemático é proporcionar uma visão geral do que tem sido investigado no contexto de geração de terrenos com voxels. Para atingir esse objetivo, seguiu-se o modelo de mapeamento sistemático proposto por Petersen, Vakkalanka e Kuzniarz (2015), onde inicialmente foram definidas 3 questões de pesquisa a serem respondidas pelo mapeamento. A seguir, as questões de pesquisa são revisitadas:

- **QP1**: Como é realizada a criação de terrenos procedurais infinitos em voxels?
- **QP2**: Quais técnicas são utilizadas na criação de terrenos em voxels?
- **QP3**: Quais tecnologias (linguagem de programação, biblioteca gráfica) são mais utilizadas na criação de terrenos em voxels e quais são seus prós e contras?

Conforme demonstrado pela carência de conteúdo sobre o assunto, não foram encontrados artigos que proporcionem uma resposta completa à QP1, o que já era esperado, dado a motivação desta pesquisa. Já para a QP2 foram encontradas algumas boas respostas a serem exploradas mais adiante, como a utilização de *chunks* para otimizar a renderização, a utilização de

hashmaps para armazenamento/manipulação de dados, além de algumas técnicas como *culling* e *greedy meshing*. Para a QP3, embora com um conjunto de estudos pequenos, foi identificada a predominância na utilização de OpenGL como API gráfica por ser multi-plataforma, gratuita e robusta. Já sobre linguagens de programação, a utilização de linguagens de conforto de cada autor demonstra que não há uma necessidade específica de uma linguagem sobre outra, porém, por possibilitar mais gerenciamento de seus recursos em baixo nível, C++ acaba sendo uma das opções mais utilizadas.

4 PROPOSTA

O objetivo deste Capítulo é apresentar a proposta da monografia. Na seção 4.1 é apresentado o problema a ser abordado, juntamente ao problema de pesquisa, a questão de pesquisa e possíveis contribuições nas Seções 4.1.1, 4.1.2 e 4.1.3 respectivamente. Por fim, os objetivos são apresentados na Seção 4.2.

4.0.1 Problema a ser abordado

Embora a utilização de voxels para gerar volume tenha muitas aplicações em diferentes áreas, no melhor do nosso conhecimento, é raro encontrar descrições completas ou exemplos amplamente documentados, disponíveis e que permitam sua utilização. Tal fato torna sua utilização limitada, especialmente por iniciantes e dificulta a realização de futuros trabalhos empregando essa tecnologia. Um ponto importante, é quando um trabalho é realizado com essa tecnologia, várias abordagens diferentes podem ser utilizadas, o que torna os trabalhos muito diferentes uns dos outros, todavia isto nem sempre fica claro na literatura disponível (LYSENKO, 2012a). Somente para citar alguns exemplos, dentre essas abordagens é possível encontrar variações nos seguintes temas: (i) difere-se muito a linguagem de programação, (ii) API de computação gráfica utilizada (OpenGL, Direct3D, Vulkan, dentre outras), (iii) além das diversas técnicas de desenvolvimento que englobam a manipulação de dados e renderização como *chunks*, *hashmaps* e *octrees*, *greedy meshing*, entre outros (LYSENKO, 2012a).

4.0.1.1 Problema de Pesquisa

No melhor do nosso conhecimento, os trabalhos acadêmicos disponíveis ou mais facilmente acessíveis neste domínio falham em fornecer um *rationale*[1], ou seja, em sua maioria não tem todos os passos documentados ou são documentados de forma *ad hoc,* o que não permite que estes estudos sejam reproduzidos ou continuados. Em síntese: (i) não documentam de forma sistemática o desenvolvimento de aplicações que façam a geração de terrenos procedurais com voxels, (ii) não disponibilizam o código fonte; (iii) em geral não deixam explícitas quais técnicas foram utilizadas ou o (iv) porque foram escolhidas.

Diebold, Vetro e Fernandez (2015) argumentam que, a transferência de tecnologia é um fator chave para o sucesso de projetos de pesquisa, especialmente em engenharia de software, onde o (real) impacto dos resultados pode depender não apenas da confiabilidade e viabilidade das tecnologias, mas também de sua aplicabilidade em ambientes industriais. Ao ignorar a documentação sistemática das etapas da investigação e o motivo das decisões tomadas, estas

[1] Um *rationale* é uma documentação explícita das razões por trás das decisões tomadas ao projetar um sistema ou artefato.

pesquisas transformam tanto os produtos como os processos investigados em “caixas-pretas”, limitando assim o avanço do conhecimento por toda a comunidade.

4.0.1.2 Questão de Pesquisa

É possível construir uma aplicação que faça a geração de terrenos procedurais “infinitos” com voxels, detalhando sua implementação sistematicamente e explicitando as técnicas utilizadas?

4.0.1.3 Possíveis Contribuições

- A criação de um protótipo que faça a geração de terrenos procedurais “infinitos” com voxels com potencial para ajudar pesquisadores que necessitam desenvolver aplicações que representam volume com voxels.
- Um trabalho acadêmico de qualidade, sistemático e que permita a reprodução do que for descoberto, podendo contribuir com a comunidade acadêmica em especial criar oportunidades para aperfeiçoamento das técnicas e tecnologias utilizadas para trabalhos futuros por outros interessados.

4.1 Objetivos

O objetivo principal deste projeto é o desenvolvimento de um protótipo que faça a geração de terreno procedural infinito com voxels, detalhando as técnicas utilizadas e definindo as abordagens utilizadas.

5 METODOLOGIA

Neste Capítulo é apresentada a metodologia proposta. O método de pesquisa utilizado é a pesquisa exploratória dada a carência de literatura de qualidade sobre o tema de pesquisa. A pesquisa exploratória visa explorar uma questão a fim de apoiar uma investigação mais precisa futuramente. Seu objetivo é se aproximar do tema, principalmente aqueles com carência de um *corpus* significativo, por meio de revisões bibliográficas e estudos de caso (DIEBOLD; VETRO; FERNANDEZ, 2015).

Na Seção 5.1 é discutida a revisão da literatura realizada, seguida pelo estudo de viabilidade das tecnologias na Seção 5.2, onde técnicas e algoritmos comumente utilizados no desenvolvimento de terreno com voxels são abordados. Por fim, na Seção 5.3 são apresentadas as considerações finais sobre a metodologia.

5.0.1 Revisão da Literatura

Inicialmente, uma breve revisão do estado da arte foi realizada e os trabalhos mais próximos da problemática foram selecionados, visando entender quais suas limitações para a definição deste trabalho.

Para selecionar trabalhos relacionados ao tema de forma mais pragmática, um mapeamento sistemático foi realizado na seção 3 conforme o processo definido por Petersen, Vakkalanka e Kuzniarz (2015), garantindo assim que os resultados possam ser replicados e principalmente, que estudos importantes pudessem ser selecionados de acordo com critérios de exclusão, a fim de responder questões de pesquisa definidas anteriormente.

Um número limitado de bases de dados foram utilizadas no processo de pesquisa, mas que retornaram uma quantidade satisfatória na seleção inicial. Porém, vários destes estudos englobam outras disciplinas, como a medicina, e outras subáreas de pesquisa, como a voxelização explicada na seção 3.

Estes resultados mostram que o desenvolvimento de terrenos com voxels pode ser uma subárea ainda não muito explorada dentro do meio acadêmico, principalmente quando o escopo é diminuído para a utilização de tecnologias específicas.

Além disso, a baixa quantidade de resultados encontrados no processo de mapeamento resultou na integração do conjunto inicial de pesquisa a fim coletar mais informações e responder às questões de pesquisa de forma mais satisfatória.

5.0.2 Tecnologias Investigadas

Para o desenvolvimento do projeto foram levantadas as principais técnicas e algoritmos encontrados que são utilizados na geração de terreno com voxels. Algumas dessas técnicas/algo-

ritmos já foram apresentados na seção 3, outras foram encontradas fora do meio acadêmico em artigos para blogs de desenvolvedores. As seguintes técnicas foram levantadas, considerando as necessidades e objetivos deste projeto em específico:

5.0.2.1 Geometry Batching e Chunking

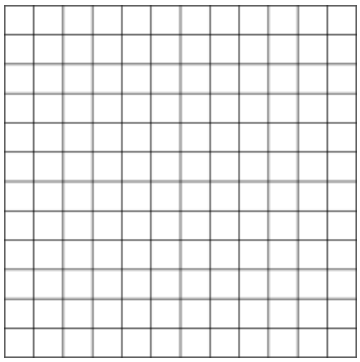
Conforme explicado com mais detalhes na Seção 3.2, a utilização de *chunks* significa, na utilização de terrenos com voxels, separar os dados de todo o terreno em partes, chamadas *chunks*, assim facilitando a manipulação e renderização dos dados do terreno. Já o processo de *geometry batching* é a simples aglomeração de dados para serem enviados à GPU em conjunto, que é possível realizar neste trabalho ao enviar os dados de todos os *chunks* a serem renderizados na tela de uma só vez ou enviar só dos chunks que sofreram alteração.

5.0.2.2 Armazenamento de Dados na Memória

Também conforme explorado na Seção 3.2, existem diferentes formas de realizar o armazenamento e consequentemente, manipulação dos dados do terreno na memória, sendo as mais comuns: *arrays*, *hashmaps* e *octrees*.

Em um *array*, todos os dados são simplesmente armazenados sequencialmente (LYSENKO, 2012a), conforme mostra a Figura 4.

Figura 4 – Demonstração teórica da estrutura de um array

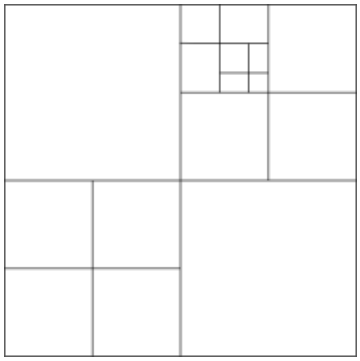


Fonte: LYSENKO (2012a).

Suas vantagens incluem facilidade de uso e bom tempo constante (não depende do tamanho dos dados) de acesso aleatório (capacidade de acessar qualquer elemento com mesma eficiência para qualquer outro) para escrita e leitura de dados. Suas desvantagens incluem o grande consumo de memória conforme o tamanho dos dados, o que significa que é apropriado para pequenos terrenos e não apropriado para os grandes e "infinitos" (LYSENKO, 2012a).

Uma *octree* é uma estrutura de árvore onde cada nó não-folha possui uma ligação com mais outros 8 nós, ou seja, ela subdivide seu espaço recursivamente em octantes de tamanhos iguais (LYSENKO, 2012a), como demonstrado na Figura 5:

Figura 5 – Demonstração teórica da estrutura de uma octree



Fonte: LYSENKO (2012a).

Suas vantagens incluem o fato de que as subdivisões de espaço encaixam muito bem para a separação de um terreno com voxels (não utilizando *chunks* nesse caso), além de um menor consumo de memória comparada à um *array* e ótimo desempenho com dados esparsos, por armazenarem apenas partes não vazias em sua estrutura. Suas desvantagens incluem a dificuldade de uso e desempenho inferior de acesso aleatório comparado ao *array* (LYSENKO, 2012a).

Um *hashmap*, ou tabela de dispersão, é uma estrutura que associa chaves e valores. Ela é comumente utilizada para manipular *arrays* de *chunks*, assim possibilitando rápido acesso apenas aos *chunks* necessários do algoritmo, facilitando também a inserção e remoção dos *chunks* de acordo com o que for necessário ser utilizado (LYSENKO, 2012a).

Suas vantagens incluem facilidade de uso, tempo constante de acesso aleatório e bom uso e armazenamento de dados esparsos. Sua desvantagem é que ocupa mais espaço do que *arrays* (LYSENKO, 2012a).

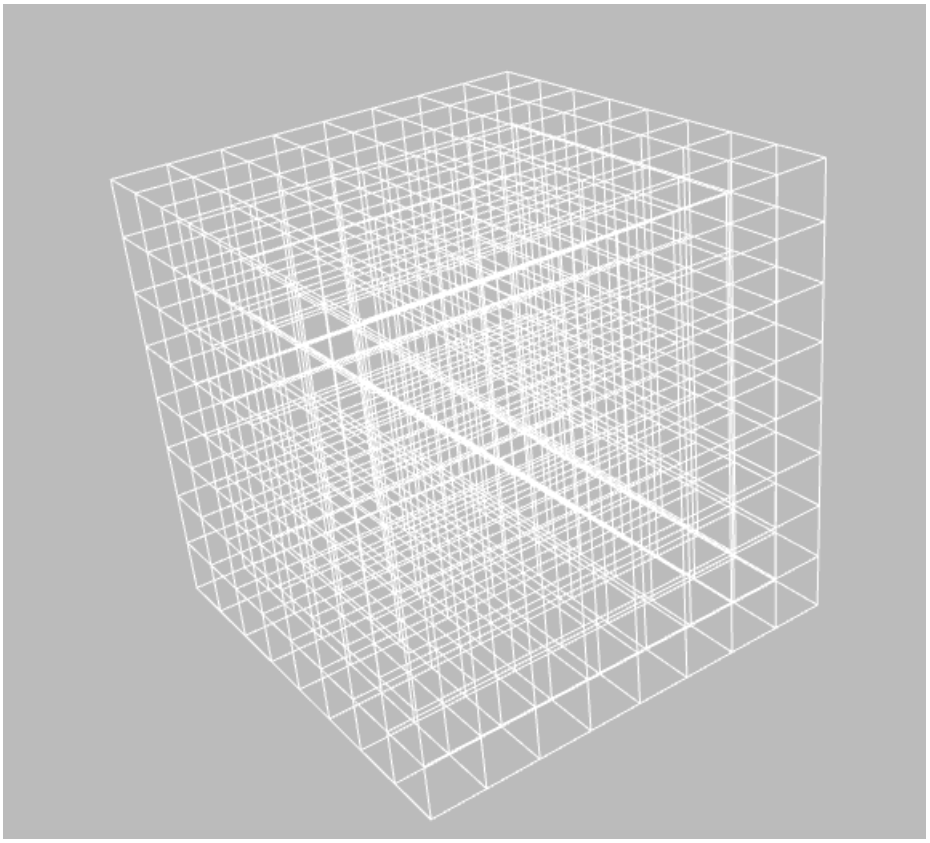
5.0.3 Face Culling

De acordo com (LYSENKO, 2012b), o jeito mais simples de gerar a malha de um *chunk* seria iterar cada voxel neste *chunk* gerando a malha de um cubo para cada, tendo complexidade de tempo linear igual ao número de voxels, conforme mostra a Figura 6.

O benefício deste método é sua simplicidade, porém, é extremamente ineficiente e de grande valia utilizar algum outro método para melhorar este caso. Neste sentido, *face culling* é o método mais adequado para melhorar em muito a geração das malhas, sem adicionar uma complexidade muito grande ao código (LYSENKO, 2012b).

Com o *face culling* na hora da iteração dos voxels em um chunk para gerar a malha, o código simplesmente precisa verificar se o voxel é visível ou não, gerando a malha apenas para os que forem visíveis. Isso é feito ao verificar para cada face do voxel, se existe um voxel sólido à sua frente, significando que aquela face não é visível por estar atrás desse voxel vizinho. A Figura 7 demonstra um exemplo de resultado deste processo (LYSENKO, 2012b).

Figura 6 – Exemplo de geração da malha de um chunk do modo simples



Fonte: LYSENKO (2012b).

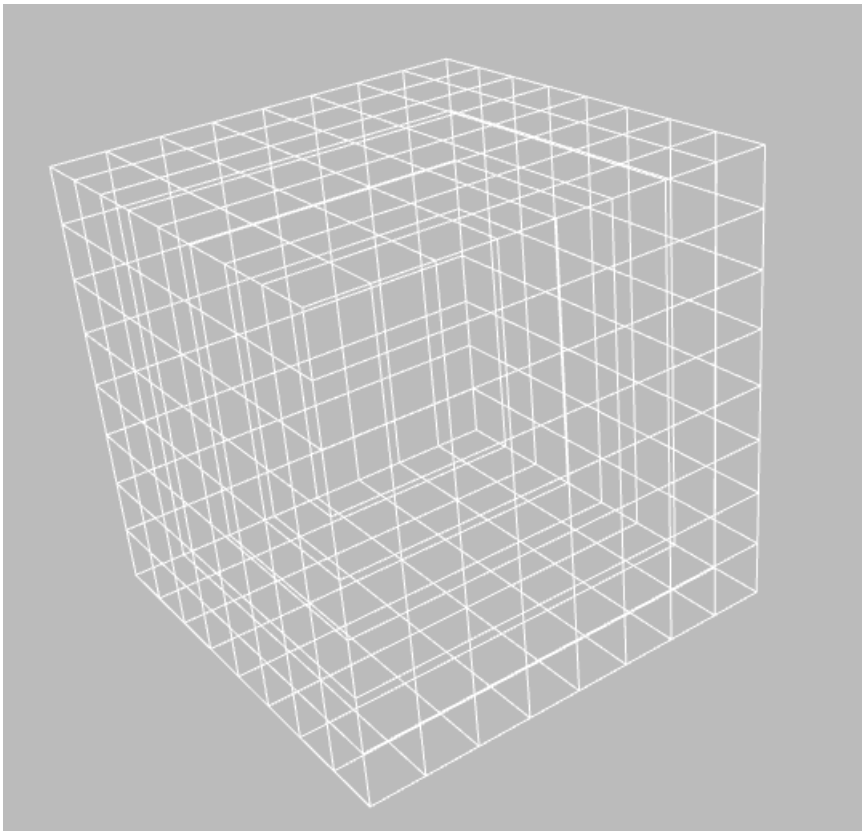
Este processo não tem uma complexidade de tempo menor que o processo anterior, pois ainda é necessário verificar os vizinhos de cada voxel além da iteração normal, o que também gera uma complexidade maior na escrita do código, pois precisa-se de alguns cuidados principalmente na verificação de voxels em chunks vizinhos. Dito isso, este processo ainda vale muito a pena pois economiza muita memória na GPU ao diminuir o tamanho da malha significantemente ((LYSENKO, 2012b).

5.0.4 Greedy Meshing

O método de *face culling* é provavelmente o mais utilizado na geração de terrenos com voxels, porém, existe ainda um método mais eficiente em alguns casos chamado de *greedy meshing*. Neste método, as malhas de voxels adjacentes que são iguais ao mesmo são mescladas, resultando em regiões maiores para reduzir o tamanho total de geometria (LYSENKO, 2012b), como demonstra a Figura 8.

Este método tem grande valor apenas em terrenos onde os voxels costumam a serem uns iguais aos outros, para que a mescla possa ser realizada. Seu maior defeito acaba sendo sua complexidade, sendo difícil balancear o tempo para gerar a malha com todas as verificações

Figura 7 – Exemplo de geração da malha de um chunk utilizando *face culling*



Fonte: LYSENKO (2012b).

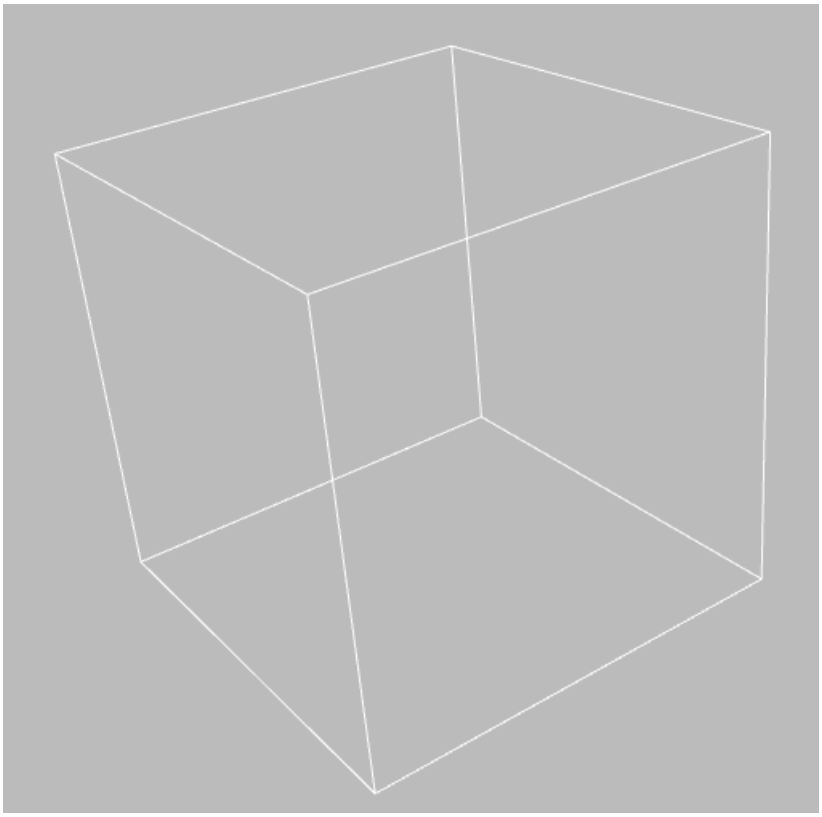
necessárias, e principalmente, a aplicação correta de texturas para cada voxel em específico caso elas sejam utilizadas, já que suas malhas individuais não existem mais (LYSENKO, 2012b).

5.0.5 Frustum Culling

Frustum, ou tronco em português, é dito por Vries (2020) como uma parte de um sólido como um cone ou uma pirâmide, usados tipicamente em aplicações gráficas para falar sobre a câmera. O *frustum* de uma câmera representa a zona de visão da mesma, onde objetos muito próximos são cortados de visão, para não dar um efeito de que você possa enxergar por dentro deles quando está muito próximo, além de cortar os objetos muito distantes, para melhorar o desempenho ao não necessitar renderizar objetos que não são foco da cena, como mostra a Figura 10.

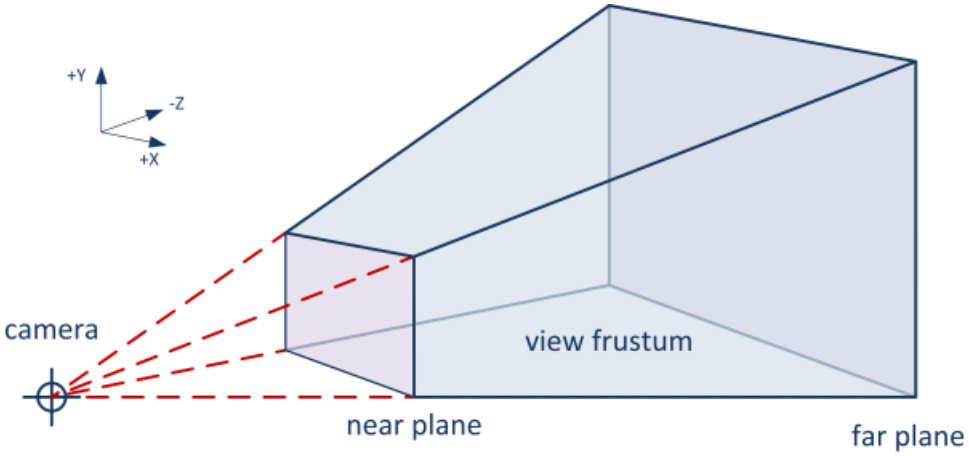
Além de cortar os objetos na hora da renderização, também faz parte do processo de *frustum culling* a adaptação do código para não necessitar cálculos de objetos que não estão dentro do *view frustum*, a fim de melhorar o desempenho (VRIES, 2020).

Figura 8 – Exemplo de geração da malha de um *chunk* utilizando *greedy meshing*



Fonte: LYSENKO (2012b).

Figura 9 – Representação do *frustum* da câmera em uma aplicação digital



Fonte: LYSENKO (2012b).

5.0.6 Multithreading

De acordo com Williams e Williams (2012), concorrência no âmbito computacional significa a capacidade de um sistema em performar múltiplas atividades em paralelo, ao invés de sequencialmente. A concorrência é limitada de acordo com o número de *threads* de um processador, que é basicamente a quantidade de quantas tarefas independentes aquele hardware consegue rodar concorrentemente.

Além da concorrência, existe ainda o conceito de paralelismo, sendo ambos parecidos. Williams e Williams (2012) explicam que ambos têm como foco a execução de tarefas em simultâneo, mas o paralelismo se preocupa mais com performance, enquanto que concorrência trata mais a separação de responsabilidades e responsividade.

As principais vantagens de um programa *multithread* então são o ganho de performance, pela possibilidade de executar tarefas em paralelo ao invés de sequencialmente, e a separação de responsabilidades, que facilita o entendimento do código e realização de testes ao separar o código que realiza tarefas diferentes. Em um código de geração de terrenos com voxels, é possível por exemplo separar a parte do código que gera as malhas do terreno do resto do código, que gerencia outras tarefas como comandos de entrada e a própria renderização (WILLIAMS; WILLIAMS, 2012).

A versão do C++11 foi a primeira a dar suporte para programas multithread, provendo componentes na biblioteca para auxiliar no desenvolvimento dos mesmos. Essa mudança possibilitou a construção de aplicações *multithread* com C++ sem depender de extensões específicas para cada plataforma, garantindo comportamento correto e portátil (WILLIAMS; WILLIAMS, 2012).

5.0.7 Algoritmos de Geração Procedural

Conforme explicado na Seção 2.3, a geração procedural é uma geração de dados que são utilizados para gerar conteúdo automaticamente. De acordo com Doran e Parberry (2010), a geração procedural de terrenos provém de um desejo de gerar este conteúdo sem necessitar gastar grandes investimentos que geralmente são requeridos para tal.

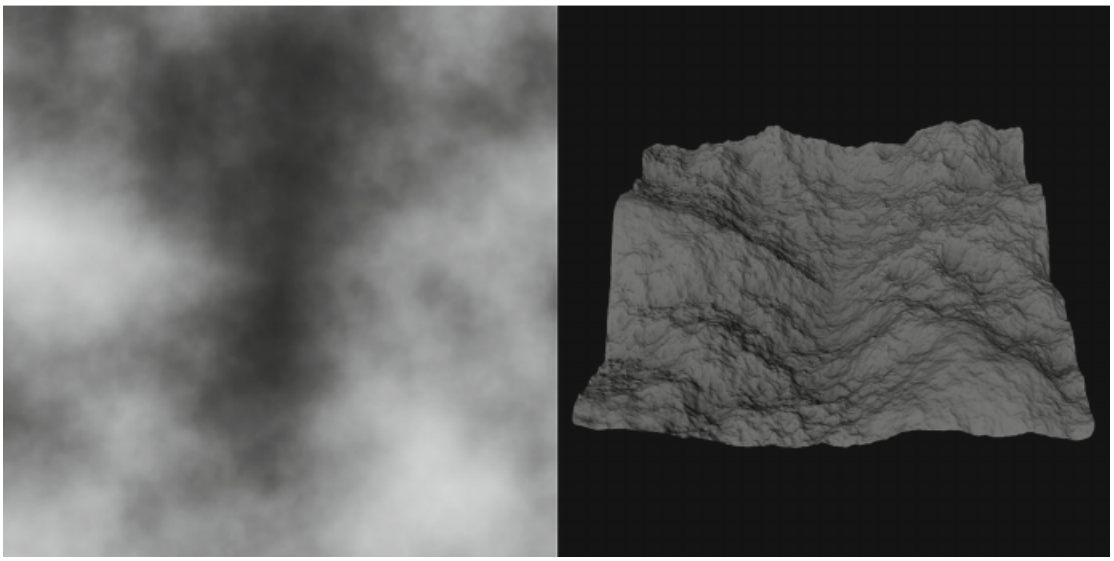
Ainda de acordo com PDoran e Parberry (2010), a geração procedural de terrenos é realizada ao gerar *heightmaps* que diferem uns dos outros de acordo com valores randômicos, chamados de *seeds*. Estes *heightmaps* são imagens que armazenam valores com relação à elevação de uma superfície, como demonstra a Figura 10.

Para que se possam ser gerados esses *heightmaps*, comumente são utilizados algoritmos já bem estabelecidos, como o *Perlin noise*, por haver uma grande complexidade na geração de algoritmos que façam esse tipo de tarefa. Com os *heightmaps* então, é possível gerar o terreno conforme a elevação do mesmo, no caso de voxels, basicamente seria necessário colocar os voxels exatamente nas posições de elevação dadas pelo *heightmap* (DORAN; PARBERRY, 2010).

5.0.8 Ambient occlusion

Ambient occlusion é uma técnica para calcular a iluminação de cada ponto de uma cena digital de acordo com uma fonte propagadora de luz ambiente, conforme explica LYSENKO

Figura 10 – Exemplo de um *heightmap* gerado com *perlin noise* (esquerda) e terreno gerado com o mesmo (direita)



Fonte: BRUMMELEN e CHEN (2018).

(2013). Basicamente, o resultado é uma simulação de sombreamento que dá um tom mais realista à cena.

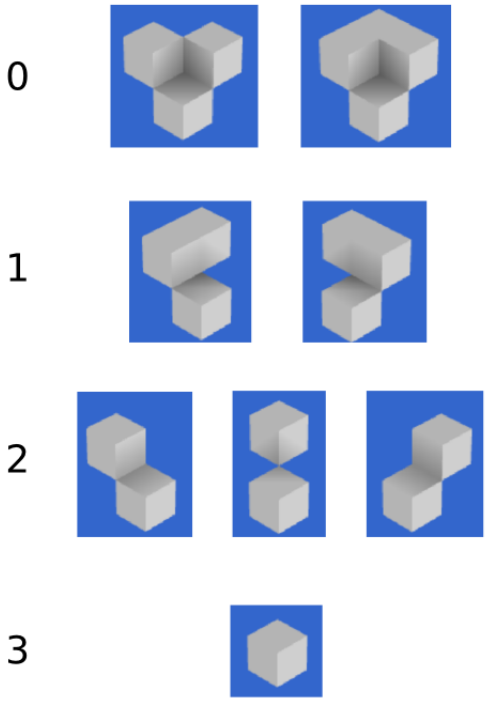
Existem diferentes modos de se calcular o *ambient occlusion*, mas para terrenos com voxels especificamente é possível fazer este cálculo de forma relativamente mais eficiente do que em outros tipos de terrenos. A ideia básica é realizar este cálculo para cada voxel apenas verificando os voxels adjacentes ao mesmo, existindo 4 tipos diferentes de resultados (LYSENKO, 2013), como demonstra a Figura 11.

5.0.9 CONSIDERAÇÕES FINAIS

Para desenvolvimento deste projeto, e devido ao tempo limitado disponível, as informações de cada técnica apresentada nesta seção foram levadas em conta a fim de avaliar sua importância para o mesmo e como resultado, as seguintes técnicas foram aplicadas no projeto:

- *Geometry Batching*
- *Chunking*
- Armazenamento de dados com *hashmap*
- *Face Culling*
- *Frustum Culling*
- Geração Procedural

Figura 11 – Representação dos 4 tipos de *ambient occlusion* para voxels



Fonte: LYSENKO (2013).

O *hashmap* foi escolhido em favor do *array* e da *octree* pelo projeto ter como foco a criação de terreno "infinito" e ter um tempo limite para sua criação, sendo que o *array* é favorável para pequenos terrenos e a *octree* mais complexa para uso.

A técnica de *greedy meshing* não foi utilizada por sua complexidade, também problemática para o tempo, além de ser especialmente útil em alguns casos e em outros não.

Já a técnica de *multithreading* foi considerada por poder favorecer no desempenho do algoritmo, porém, seu desenvolvimento se apresentou complexo por necessitar de uma grande reestruturação na arquitetura do código, já que na estrutura apresentada a modificação e renderização de *chunks* estão interligadas fortemente a fim de simplificar o algoritmo. Para aplicar *multithreading*, de forma simplificada e teórica, seria necessário armazenar *chunks* prontos para serem renderizados em uma nova estrutura que é utilizada por uma *thread* de renderização, sendo que outra *thread* trabalharia na atualização de *chunks* de forma independente, inserindo os *chunks* atualizados na estrutura dos prontos apenas após finalizar seu trabalho nos mesmos.

Por fim, a técnica de *ambient occlusion* também foi considerada, e embora tenha uma aplicação mais fácil em projetos que utilizam voxels, ainda pode ser considerada complexa de se implementar quando não há muita experiência com *shaders* e OpenGL. Além disso, a aplicação de algoritmos de iluminação básica como *ambient* e *diffuse lighting* é realizada de forma fácil, pois é a mesma em qualquer projeto em OpenGL, e diminuem a necessidade da técnica de *ambient occlusion* para auxiliar na visibilidade dos voxels.

6 DESENVOLVIMENTO

Neste Capítulo são relatados de forma gradual os passos para o desenvolvimento do algoritmo. Primeiramente, a Seção 6.1 tem como objetivo explicar quais partes do algoritmo final não serão apresentadas por estarem fora de escopo, a Seção 6.2 define as tecnologias utilizadas, para então na Seção 6.3 começar a explicação dos passos com a criação de um *chunk*. Seguindo, a Seção 6.4 apresenta a criação de *chunks* de forma "infinita", a Seção 6.5 trata da aplicação da técnica de *face culling*, a Seção 6.6 detalha o processo de aplicação de um algoritmo de geração procedural e a Seção 6.7 sobre a técnica de *frustum culling*. Por fim, a Seção 6.8 apresenta as considerações finais de desenvolvimento.

6.0.1 O QUE NÃO SERÁ ABORDADO

Algumas partes do algoritmo criado e apresentado na seção 6 serão pouco ou não abordadas, pois são códigos gerais não específicos ao escopo deste projeto, ou seja, são de conhecimento geral e uso comum, encontrados na literatura como por exemplo no livro de (VRIES, 2020). Os seguintes algoritmos não serão apresentados:

- Integração de bibliotecas e uso da biblioteca "ImGui" para *debug* do projeto;
- Criação de janela e configuração do OpenGL;
- Algoritmo de câmera com movimentação 3D;
- *Shaders* e iluminação básica, especificamente *ambient* e *diffuse lighting*;
- Sistemas de coordenadas;

A Seção 6 então tem como foco a apresentação e explicação de algoritmos específicos para a resolução do problema proposto, ou seja, a criação de um terreno procedural "infinito" em voxel.

6.0.2 TECNOLOGIAS UTILIZADAS

Para desenvolvimento, a linguagem de programação C++ foi escolhida por possibilitar manipulação direta de recursos e possuir a maior quantidade de recursos para estudo disponíveis, conforme mencionado na Seção 3.4. Ainda de acordo com a Seção 3.3, a API de computação gráfica escolhida foi o OpenGL em sua versão 3.3, a qual será utilizada para o renderizamento de imagens, por possibilitar desenvolvimento multi-plataforma e também ter a maior quantidade de recursos para estudo disponíveis.

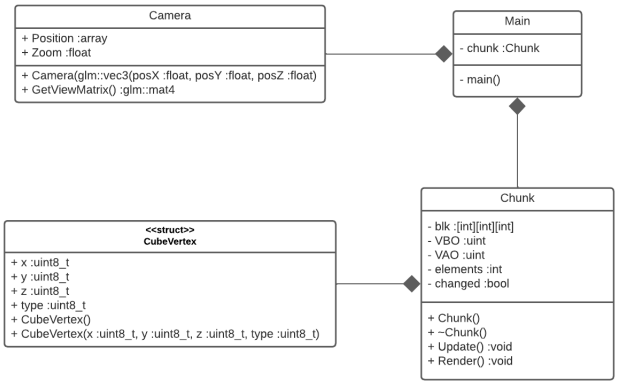
É importante notar que no momento em que este trabalho é escrito existem versões mais novas do OpenGL, porém, a versão 3.3 é a base de utilização de seu novo modo de desenvolvimento, o *core-profile*, conforme explicado na Seção 2.2. Embora versões mais novas do OpenGL possuam novas funcionalidades, não necessariamente é necessário utilizar alguma delas para a realização deste projeto e nenhuma delas modifica o funcionamento geral do OpenGL, além do fato principal de que a versão 3.3 é mais acessível às GPUs mais antigas.

6.0.3 CRIAÇÃO DE UM *CHUNK*

Conforme explicado na Seção 3.2 e 5.2.1, *chunks* são as partes de todo o volume do terreno, sendo compostos de vários voxels. Sendo o objetivo a criação de um terreno "infinito", a estruturação e criação de um *chunk*é a primeira etapa a ser alcançada para que então possamos realizar a criação de vários chunks em diferentes posições, gerando assim o terreno como um todo (LYSENKO, 2012a)

O diagrama UML na Figura 12 apresenta os objetos, atributos e métodos necessários para se atingir a criação de um *chunk*.

Figura 12 – Diagrama de classe UML da criação de *chunk*



Fonte: Autoria própria.

Para o desenvolvimento do algoritmo deste projeto, representamos o objeto de uma classe "Chunk" com os seguintes atributos:

- Um *array* de 3 dimensões de valores inteiros, representando a matriz 3D de voxels do *chunk* que armazena valores especificando o tipo do voxel naquela posição, incluindo o valor zero significando a inexistência de um voxel;
- VBO e VAO necessários para manipulação de memória do *chunk* no OpenGL, conforme descrito na Seção 2.2.2;
- Uma variável de inteiros acompanhando a quantidade de voxels existentes no *chunk*, apenas para auxílio nos métodos;

- Uma variável *booleana* especificando se houve alteração no *chunk*, significando a necessidade de refazer a malha do mesmo.

Conforme representado no diagrama, o algoritmo inicia na função "Main" que faz a criação de um objeto "Chunk". O código do *constructor* da classe "Chunk" é apresentado no código 1, onde acontece a alocação de memória e inicialização de atributos da classe, além de um loop triplo para definir a existência de voxels em todas as posições do *chunk*, tendo as variáveis CX, CZ e CY como configurações de sistema para delimitar a quantidade de voxels em cada lado de um *chunk*.

Listagem 1 – Constructor da classe "Chunk"

```
1  Chunk::Chunk() {
2      memset(blk, 0, sizeof(blk)); // Aloca memória para os voxels
3      elements = 0; // Define a quantidade de voxels como sendo zero
4      changed = true; // Define que o chunk precisa refazer sua malha
5      glGenVertexArrays(1, &VAO); // Criação do componente VAO
6      glGenBuffers(1, &VBO); // Criação do componente VBO
7      for (int x = 0; x < CX; x++)
8          for (int y = 0; y < CZ; y++)
9              for (int z = 0; z < CZ; z++)
10                 blk[x][y][z] = 1; // Define a existência de voxels em todas as
11 }
```

Fonte: Autoria própria (2023).

Em sequência, a função "Main" entra no loop principal da aplicação, um loop que só acaba quando o programa é finalizado, onde são constantemente processadas as entradas do usuário (movimentação da câmera) e gerenciado a renderização do programa. Dentro deste loop é chamado o método de renderização "Render" do objeto "Chunk", o qual gerencia a renderização do mesmo. Dentro deste método, é necessário verificar se o objeto foi modificado para chamar seu método de "Update". Após, é verificado se ele possui voxels para renderizar, habilitando componentes do OpenGL e o VAO do objeto para chamar a função de renderização. Essas aplicações são demonstradas no código 2.

Para que o processo de renderização deste *chunk* esteja completo, é necessário por fim a definição de seu método Update onde a criação da malha de cada voxel acontece, sendo necessário para todo *chunk* recém criado ou modificado.

A malha dos voxels criadas é feita com polígonos, que conforme LYSENKO (2012b), é uma das inovações utilizadas pelos populares jogos "Minecraft" e "Infiniminer", sendo assim, cada lado de um voxel é representado por dois triângulos que juntos formam um quadrado.

É importante ressaltar que, conforme figura 12, uma *struct* com tipagem chamada de "CubeVertex" foi criada para auxiliar no armazenamento do vértice de cada voxel do *chunk*. Além disso, todos os vértices de todos os *voxels* do *chunk* são armazenados em um array para no fim serem enviados em conjunto à GPU, no processo apresentado nas Seções 3.2 e 5.2.1 chamado de *geometry batching*.

Listagem 2 – Método “Render” da classe “Chunk”

```
1  void Chunk::Render() {
2      // Caso o chunk tenha sido modificado, é necessário chamar seu
3      método de Update para recriar sua malha
4      if (changed)
5          Update();
6
7      // Se o chunk não possuir voxels, não há nada para renderizar
8      if (!elements)
9          return;
10     // Por fim, são habilitados componentes do OpenGL e o VAO
11     especifico deste chunk para chamar a função de renderização
12     glEnable(GL_CULL_FACE);
13     glEnable(GL_DEPTH_TEST);
14     glEnable(GL_MULTISAMPLE);
15
16     glBindVertexArray(VAO);
17
18     glDrawArrays(GL_TRIANGLES, 0, elements);
19 }
```

Fonte: Autoria própria (2023).

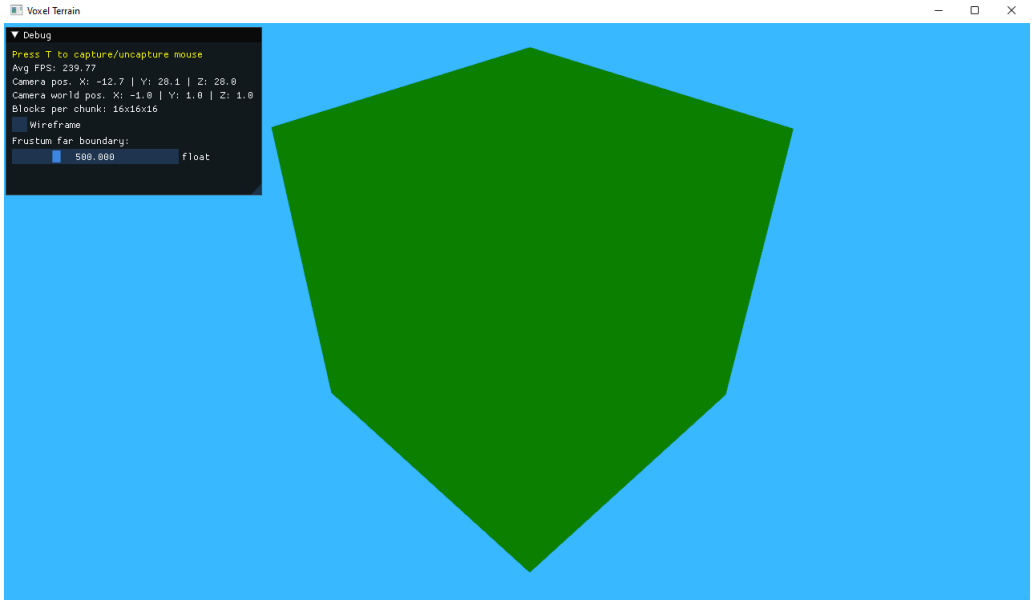
O código 3 apresenta a aplicação do método "Update" da classe "Chunk", começando com seu atributo que define se há alterações tendo seu valor mudado para falso, assim evitando que este *chunk* entre neste método novamente antes de ser modificado. Em sequência, um array é criado onde são armazenados dados dos vértices, processo este realizado na sequência em um loop de 3 dimensões criado para acessar todos os voxels deste *chunk*, criando a malha dos triângulos que formam cada lado de cada voxel. Por fim, o número de elementos deste *chunk* é atualizado e componentes do OpenGL são utilizados para definir atributos e enviar estes dados para a GPU.

Com a malha do chunk criada ou atualizada no método "Update", a renderização do *chunk* pode ser realizada tendo os dados das malhas na GPU e os atributos no *shader*. A Figura 13 mostra o resultado da criação do *chunk*, já a Figura 14 mostra os vértices do *chunk* criados com a opção de renderização *wireframe*.

6.0.4 CRIAÇÃO DO TERRENO

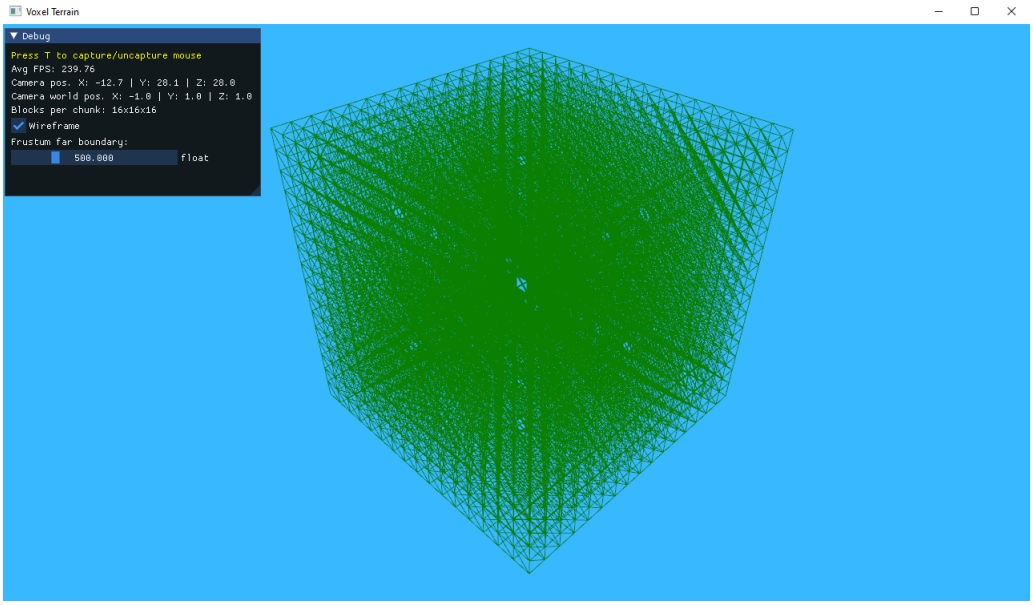
Conforme explicado nas Seções 3.2, 5.2.1 e na seção anterior, a técnica de *chunks* tem como objetivo a separação do terreno em fragmentos, chamados de *chunks*. Sendo assim, é possível criar e renderizar um terreno ao todo com a combinação dos *chunks* que o formam, porém, de acordo com LYSENKO (2012a), dependendo do tamanho do terreno, sua geração e renderização como um todo pode não ser possível de uma só vez.

Figura 13 – Resultado da criação de um *chunk* no algoritmo



Fonte: Autoria própria (2023).

Figura 14 – Resultado da criação de um *chunk*no algoritmo com *wireframe*



Fonte: Autoria própria (2023).

A utilização de *chunks* então é uma das soluções deste problema, pois, tendo o terreno já separado em *chunks*, é possível gerar e renderizar apenas os *chunks* que estão próximos à câmera da aplicação. Então, conforme a câmera se move, a aplicação pode destruir chunks que ficaram muito longe e gerar e renderizar novos *chunks* que agora estão próximos à câmera. No caso de um terreno "infinito", bastaria continuar gerando novos *chunks* conforme a câmera se move, tendo como limite apenas o espaço de armazenamento dos mesmos (LYSENKO, 2012a)).

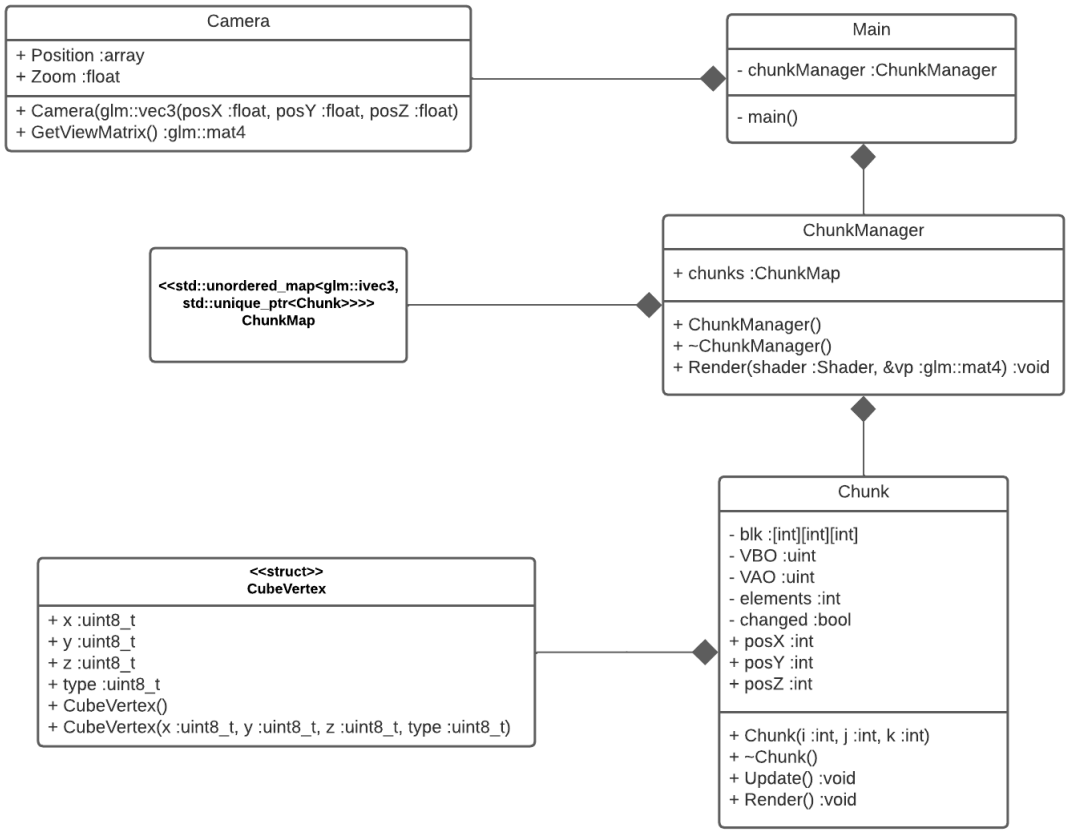
Conforme abordado na Seção 5.2.2.3, a utilização de *hashmaps* para o armazenamento e manipulação dos *chunks* então facilita todo este trabalho, pois tendo como chave a posição do

chunk e como valor o *chunk* em si, o acesso de chunks com base na posição da câmera é feito de forma direta e com tempo de resposta O(1).

6.0.4.1 Criação de Vários *Chunks*

Para realizar a criação do terreno "infinito" deste projeto, a primeira etapa então é estabelecer uma nova classe para gerenciamento dos *chunks* e então realizar a geração e renderização de vários deles. A Figura 15 apresenta a nova composição da estrutura do projeto com esta nova classe de gerenciamento de *chunks* chamada de "ChunkManager", composta pelo atributo de *hashmap* para armazenamento dos *chunks* e o método "Render", que faz a chamada para a renderização de vários *chunks*.

Figura 15 – Diagrama de classe UML da criação de vários *chunks*



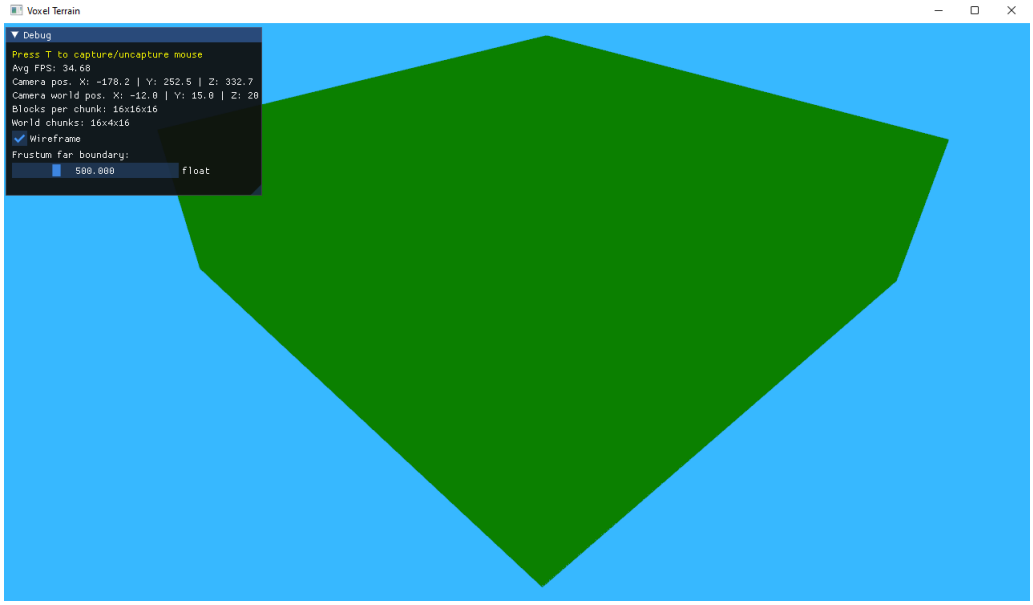
Fonte: Autoria própria (2023).

Onde antes era a criação de um objeto "Chunk", agora no novo fluxo é realizada a criação de um objeto "ChunkManager" dentro da função "main". O código abaixo apresenta o construtor da classe "ChunkManager", onde a posição da câmera é obtida para que se possa fazer um loop, criando chunks armazenados no hashmap da "ChunkManager" . É importante notar que as variáveis SCX, SCY e SCZ são configurações do projeto e definem a quantidade de chunks a serem criados, conforme ilustrado no código 4.

Com a criação do objeto da classe "ChunkManager" no lugar antes do objeto "Chunk", também é necessário modificar a chamada do método "Render" dentro do loop principal da função "main", agora fazendo a chamada para o objeto da classe "ChunkManager". O código a seguir apresenta o método "Render" desta classe, onde é realizada a iteração entre os chunks existentes armazenados no hashmap, utilizando da matriz "model" para renderizar o chunk atual em sua posição correta, podendo então finalmente chamar seu próprio método de "Render" que realiza a renderização do mesmo, conforme é apresentado no código 5.

Com a utilização da classe "ChunkManager" apresentada, vários *chunks* são criados em volta da posição inicial da câmera. A Figura 16 abaixo apresenta o resultado alcançado.

Figura 16 – Resultado da criação de vários *chunks* no algoritmo com *wireframe*



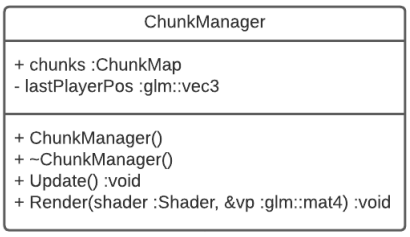
Fonte: Autoria própria (2023).

6.0.4.2 Criação de *Chunks* "Infinitos"

A criação de vários *chunks* e utilização da classe "ChunkManager" facilita agora o objetivo principal da criação de *chunks* "infinitos", sendo necessário agora fazer a criação ou destruição de chunks de acordo com sua posição relativa à câmera, conforme ela se move. Para isso, a classe "ChunkManager" necessita de um método "Update" onde isso será realizado. A Figura 17 apresenta as mudanças na classe do "ChunkManager".

Dentro do método Update é necessário obter a posição da câmera atual, então realizar uma iteração entre os chunks, verificando se estão muito distantes da câmera e os excluindo. Após, itera-se novamente para realizar a criação de chunks em posições próximas à câmera e que deveriam existir, mas não existem ainda.

Figura 17 – Diagrama UML da classe “ChunkManager” para *chunks* “infinitos”



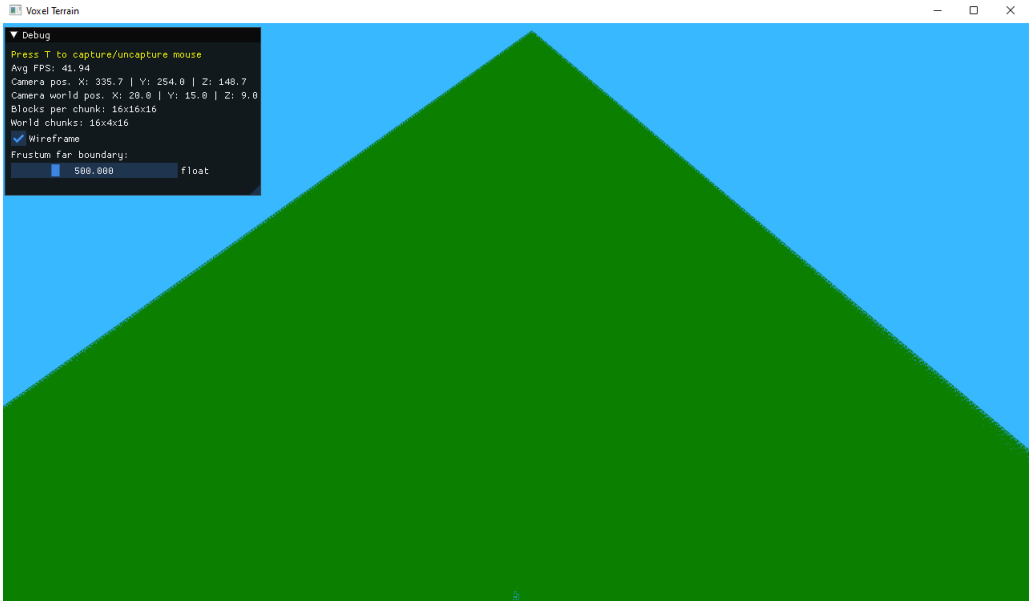
Fonte: Autoria própria (2023).

É importante notar que a distância estabelecida onde chunks devem existir ou não depende ainda das constantes SCX, SCY e SCZ. O código 6 é o utilizado no projeto para este método.

Com o método Update implementado, basta agora atualizar o método "Render" da classe "ChunkManager", verificando se a posição da câmera mudou em relação à última chamada deste método, e em caso positivo, chama-se o método "Update". A atualização é apresentada no código 7.

Com estas alterações, *chunks* próximos agora são criados e chunks distantes são excluídos de acordo com a posição da câmera, criando-se um efeito de terreno "infinito", limitado apenas pela memória da aplicação. A Figura 7 abaixo demonstra o resultado alcançado.

Figura 18 – Resultado da criação de *chunks* “infinitos” no algoritmo com *wireframe*



Fonte: Autoria própria (2023).

6.0.5 APLICAÇÃO DE *FACE CULLING*

Conforme descrito na Seção 6.3, LYSENKO (2012b) define a utilização de polígonos para a geração da malha dos voxels como uma forma inovadora de realizar a representação deste

volume, porém, com o desafio de como realizar isto de forma eficiente, sendo uma das soluções abordadas o método de *face culling*, já apresentado na Seção 5.2.3.

Ainda de acordo com LYSENKO (2012b), dois critérios importantes precisam ser levados em consideração na criação da malha de voxels/chunks, sendo eles:

- Malhas menores são malhas melhores
- O tempo de criação das malhas não pode ser muito alto

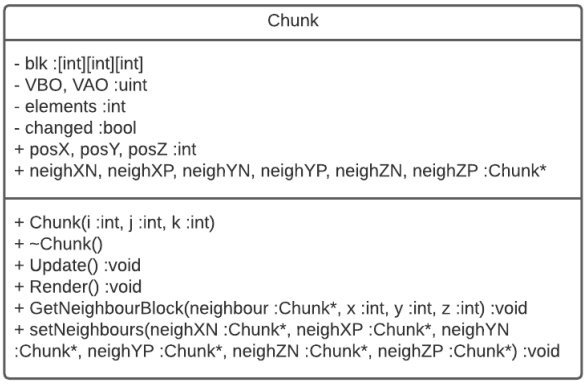
Estes critérios apresentam o principal problema na criação de malhas: velocidade vs qualidade. Ainda de acordo com LYSENKO (2012b), alguns algoritmos podem focar em qualidade, porém estes pagam o preço com uma responsividade ruim, já os algoritmos que focam em velocidade tem como detrimento a qualidade.

O *face culling* é uma técnica que, de forma geral, traz uma solução para a criação de malhas que não tem como foco nem velocidade nem qualidade, mas sim uma solução mediana para ambos, atendendo os critérios de LYSENKO (2012b) com um tempo de criação de malhas pequeno com malhas não muito grandes.

Até então, este projeto fez a criação das malhas dos voxels/chunks da forma mais simples, gerando uma malha para cada voxel existente, mesmo os não visíveis. O objetivo então com a técnica de *face culling* é realizar a geração apenas da malha dos lados visíveis dos voxels, assim economizando muita memória, com um custo pequeno para realizar as verificações de quais lados estão visíveis.

Neste projeto, a técnica de *face culling* é implementada no método Update da classe Chunk, onde a malha de cada *chunk* é gerada. É importante lembrar que para realizar as verificações necessárias sobre a visibilidade dos voxels do chunk, é necessário também o acesso aos chunks vizinhos do mesmo, pois os voxels localizados nos extremos do chunk precisam ser comparados com os voxels no outro extremo do chunk vizinho. A Figura 19 a seguir demonstra as alterações na classe "Chunk", agora com métodos auxiliares para definir *chunks* vizinhos e acessar voxels de *chunks* vizinhos.

Figura 19 – Diagrama UML da classe “Chunk” para *face culling*



Fonte: Autoria própria (2023).

O método "GetNeighbourBlock" recebe como parâmetro um chunk e três variáveis que definem posição, verificando a existência de um voxel nessa posição e o retornando, ou retornando zero caso ele não exista. Sua implementação é apresentada no código 8.

Já o método "setNeighbours" recebe como parâmetro os ponteiros para cada um dos seis *chunks* vizinhos, salvando seus valores no *chunk* atual caso eles tenham mudado, conforme implementação apresentada no código 9.

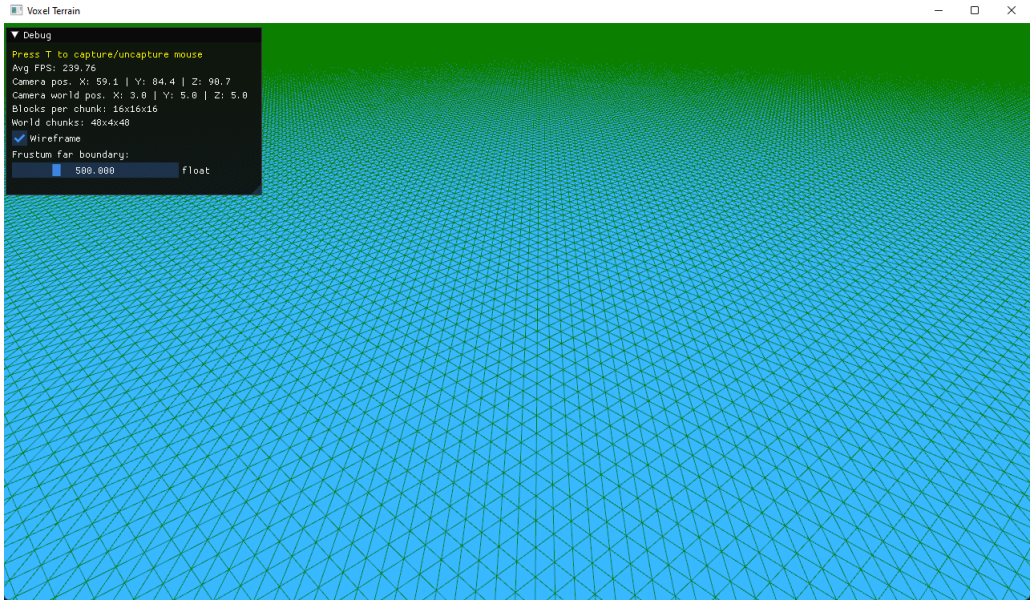
Com os métodos auxiliares definidos, é necessário agora definir os *chunks* vizinhos de cada *chunk*, informação importante para implementar esta técnica. Isso começa no método "Update" da classe "ChunkManager", onde é necessário iterar mais uma vez sobre os chunks existentes, chamando o método auxiliar "setNeighbours" para definir seus *chunks* vizinhos. O código desta alteração é apresentado no código 10.

Tendo informações dos *chunks* vizinhos e os métodos auxiliares, agora é possível realizar a implementação da técnica, que é uma simples verificação antes de realizar a criação da malha de cada lado de um voxel. Dentro do método "Update" da classe "Chunk", antes da criação da malha de cada lado de um voxel, é verificado se **não** existe um voxel visível em sua frente, para apenas assim criar a malha.

Como apresentado anteriormente nesta seção, o objetivo desta alteração é não criar as malhas dos lados de voxels que não estão visíveis, pois existem outros voxels à sua frente. A alteração no código é apresentada no código 11, sendo que a verificação é realizada para todos os lados do voxel.

O resultado desta técnica é apresentado na Figura 20, onde percebe-se a diminuição da malha do terreno, existindo apenas em seu lado visível.

Figura 20 – Resultado da aplicação de *face culling* no algoritmo com *wireframe*



Fonte: Autoria própria (2023).

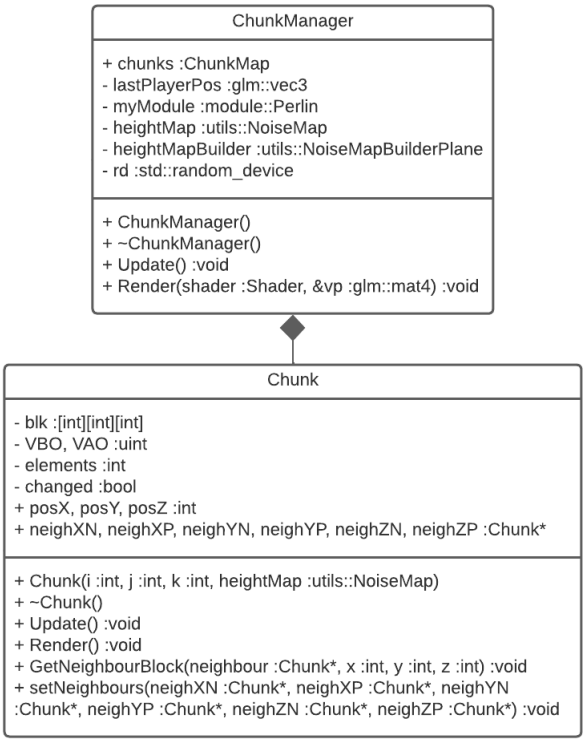
6.0.6 APLICAÇÃO DE GERAÇÃO PROCEDURAL

Segundo abordado na seção 5.2.7, algoritmos de geração procedural como o Perlin Noise são algoritmos que geram dados e os armazenam em mapas de elevação chamados *heightmaps*. Estes mapas são gerados utilizando um número chamado de *seed*, onde para cada *seed* dados diferentes são gerados. Além disso, estes *heightmaps* geram valores contínuos, ou seja, em um *heightmap* de tamanho x, é possível mudar sua posição e gerar os dados de x + 1, dos quais possuem continuidade com os dados de x e não gerando alterações bruscas em suas extremidades (CNT, 2003).

Para facilitar a utilização desta técnica, existem bibliotecas prontas e de código aberto para várias linguagens, sendo a *libnoise* uma das opções para C++ e a utilizada neste projeto (CNT, 2003). Dito isso, a aplicação do algoritmo no projeto é direta e com as informações abordadas aqui, a utilização de outro algoritmo pode ser considerada.

Para a utilização do *libnoise*, uma mudança na estrutura das classes "ChunkManager" e "Chunk" é necessária, adicionando atributos para armazenamento e geração de um *heightmap*. Além disso, também é introduzido um atributo da classe padrão do C++ para auxiliar na geração de valores aleatórios, importantes para definir o valor do *seed*. Esta mudança é apresentada na Figura 21.

Figura 21 – Diagramas UML das classes "ChunkManager" e "Chunk" para geração procedural



Fonte: Autoria própria (2023).

Com a nova estrutura destas classes, a primeira etapa é gerar um *heightmap* quando um objeto "ChunkManager" é criado, pois os dados do *heightmap* serão utilizados para a criação de cada *chunk* neste objeto. Utilizando o atributo que auxilia na criação de um valor aleatório,

o valor do *seed* é definido para utilização na geração do *heightmap*, ou seja, toda vez que o projeto for aberto, dados diferentes serão gerados para a criação do terreno. Estas alterações são apresentadas no código 12 do constructor da classe "ChunkManager".

Com o atributo do *heightmap* criado e armazenado na classe "ChunkManager", o necessário agora é passar os dados do *heightmap* na criação de cada *chunk*, para que então seja possível definir os valores de cada voxel. Para isso, o método "Update" da classe "ChunkManager" é alterado, utilizando limites diferentes para cada *chunk* e então gerando dados com o *heightmap* criado, por fim passando esses dados em sua construção.

É importante ressaltar que antes da geração de dados com o *heightmap*, seus limites precisam ser alterados de acordo com a posição do *chunk* a ser criado, para que assim seja garantida a continuidade dos dados de cada *chunk*. A alteração no código é apresentada no código 13.

Com os dados do *heightmap* corretos, contínuos e de tamanho igual ao do *chunk* sendo passados ao mesmo, basta utilizá-los para definir até que altura devem ser criados voxels em cada posição, pois são dados de elevação. Para isto, é necessário atualizar o *constructor* da classe "Chunk", recebendo os valores do *heightmap* e modificando o *loop* de criação dos voxels, realizando o loop apenas para os valores das posições X e Z, como se estivesse trabalhando em um grid 2D. Dentro deste *loop*, o valor do *heightmap* na posição X e Z é obtido, sendo este valor o que define a elevação nesta posição, ou seja, se o valor é 5, voxels devem ser criados nesta posição até atingir o valor de altura (Y) de 5.

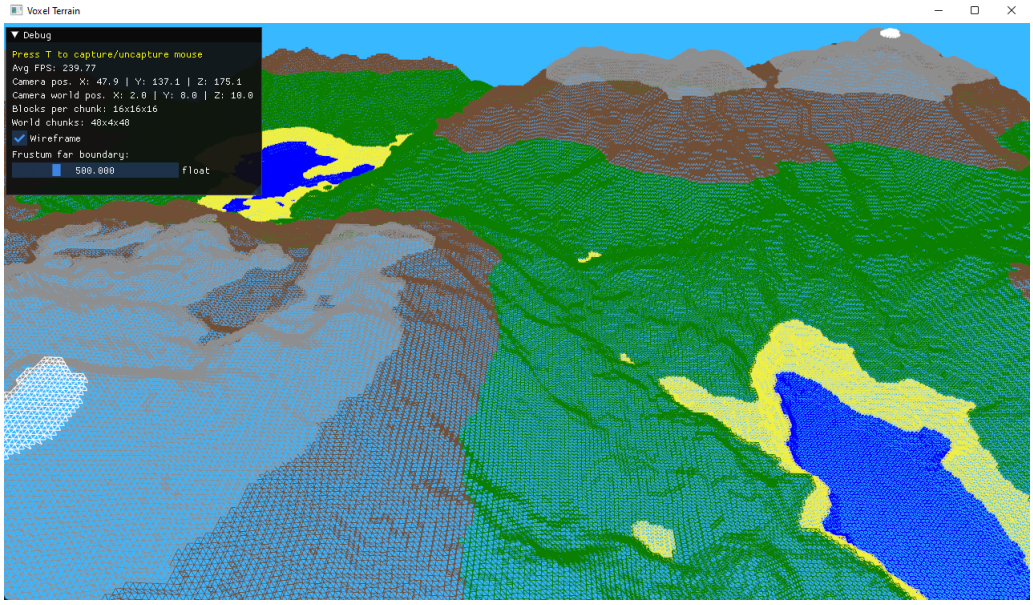
Por fim, para facilitar a visualização destes voxels, seus valores são definidos de acordo com sua posição global utilizando *enums* que descrevem sua coloração, ou seja, voxels em posições muito altas em escala global recebem um valor *enum* do tipo "BlockType_Snow", que é utilizado no *shader* para renderizar este bloco na coloração branca. O código alterado do *constructor* é apresentado no código 14.

A Figura 22 apresenta os resultados obtidos com a implementação desta técnica, onde o terreno obteve formato com continuidade entre os *chunks*, além da coloração de acordo com altura para facilitar na visualização.

6.0.7 APLICAÇÃO DE *FRUSTUM CULLING*

Conforme apresentado na seção 5.2.5, *frustum culling* é uma técnica que envolve o corte e não renderização de objetos muito distantes, além de evitar cálculos de objetos que não estejam dentro do raio da câmera. Quando há separação do terreno com *chunks* como utilizado neste projeto, a implementação de *frustum culling* é facilitada, pois é possível simplesmente mudar a quantidade de *chunks* a serem criados, diminuindo a distância de renderização do terreno e cálculos do mesmo.

Figura 22 – Resultado da aplicação de geração procedural no algoritmo com *wireframe*



Fonte: Autoria própria (2023).

A única alteração necessária no projeto então para aplicação desta técnica é não realizar o cálculo e renderização de *chunks* que estão dentro do raio de distância definido, mas não no ângulo da câmera, ou seja, *chunks* que estão atrás da câmera e não visíveis naquele momento.

Para aplicação desta alteração, o método "Render" da classe "ChunkManager" é alterado, onde é necessário verificar se a posição do *chunk* está dentro do raio de visão da câmera, e caso não esteja, este *chunk* é pulado e seu método "Render" e consequentemente "Update" não são chamados, evitando cálculos e renderizações desnecessárias. A alteração no código é apresentada no código 15.

Com esta simples alteração o projeto é otimizado, melhorando seu desempenho de forma considerável.

Listagem 3 – Método "Update" da classe "Chunk"

```
void Chunk::Update() {
    // Define o atributo de modificado como falso, pois o chunk está
    sendo atualizado agora
    changed = false;
    // Cria um array para armazenar os dados de todos os vértices dos
    voxels CubeVertex vertex[CX * CY * CZ * 6 * 6];
    int i = 0;
    // Loop de 3 dimensões para atingir todos os voxels
    for (int x = 0; x < CX; x++) {
        for (int y = 0; y < CY; y++) {
            for (int z = 0; z < CZ; z++) {
                uint8_t type = blk[x][y][z];
                // Caso não exista um voxel nesta posição, não há nada
                para ser feito
                if (!type)
                    continue;
                // Caso exista, é necessário definir os vértices de
                cada triângulo que compõe cada lado de um voxel
                // LADO X NEGATIVO DO VOXEL
                // triângulo de baixo
                vertex[i++] = CubeVertex(x, y, z, type);
                vertex[i++] = CubeVertex(x, y, z + 1, type);
                vertex[i++] = CubeVertex(x, y + 1, z, type);
                // triângulo de cima
                vertex[i++] = CubeVertex(x, y + 1, z, type);
                vertex[i++] = CubeVertex(x, y, z + 1, type);
                vertex[i++] = CubeVertex(x, y + 1, z + 1, type);
                // LADO X POSITIVO DO VOXEL
                            ...
            }
        }
    }
    // Atualiza o número de voxels do chunk
    elements = i;

    // Habilita o VAO e VBO para então armazenar os vértices na GPU
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);

    glBufferData(GL_ARRAY_BUFFER, elements * sizeof(*vertex), vertex,
    GL_STATIC_DRAW);

    // Define o atributo de posição no shader
    glVertexAttribPointer(0, 3, GL_BYTE, GL_FALSE, sizeof(CubeVertex),
    (void*)0);
    glEnableVertexAttribArray(0);
    // Define o atributo de tipo no shader
    glVertexAttribIPointer(1, 1, GL_BYTE, sizeof(CubeVertex), (void*)
    (3 * sizeof(uint8_t)));
    glEnableVertexAttribArray(1);
}
```

Fonte: Autoria própria (2023).

Listagem 4 – *Constructor* da classe "ChunkManager"

```
ChunkManager::ChunkManager() {
    glm::vec3 key;
    // Obtêm a posição da c mera e a armazena em variáveis auxiliares
    int playerPosX = floor(camera.Position[0] / 16);
    int playerPosY = floor(camera.Position[1] / 16);
    int playerPosZ = floor(camera.Position[2] / 16);
    // Loop triplo em volta da posição da c mera
    for (int x = playerPosX + (-SCX / 2); x <= playerPosX + SCX / 2;
    x++)
        for (int y = playerPosY + (-SCY /
        2); y <= playerPosY + SCY / 2; y++)
            for (int z = playerPosZ + (-SCZ / 2); z <= playerPosZ +
            SCZ / 2; z++) {
                // Criação de um chunk, armazenando-o no hashmap
                key.x = x; key.y = y; key.z = z;
                chunks[key] = std::unique_ptr<Chunk>(new Chunk(x, y,
                z));
            }
}
```

Fonte: Autoria própria (2023).

Listagem 5 – Método "Render" da classe "ChunkManager"

```
void ChunkManager::Render(Shader shader, glm::mat4 &vp) {
    // Itera entre os chunks existentes caso exista algum
    glm::vec3 pos;
    if (!chunks.empty()) {
        for (ChunkMap::iterator iter = chunks.begin();
        iter != chunks.end(); ++iter) {
            // Utiliza da matriz "model" para renderizar o chunk atual
            em sua posição correta
            pos.x = iter->second->posX; pos.y = iter->second->posY;
            pos.z = iter->second->posZ;
            glm::mat4 model = glm::translate(glm::mat4(1), pos);
            shader.setMat4("model", model);
            // Faz a chamada para renderização do próprio método
            "Render" do chunk
            iter->second->Render();
        }
    }
}
```

Fonte: Autoria própria (2023).

Listagem 6 – Método "Update" da classe "ChunkManager"

```
1  void ChunkManager::Update() {
2      glm::vec3 key;
3      // Obtêm a posição da c mera e a armazena em variáveis auxiliares
4      int playerPosX = floor(camera.Position[0] / CX);
5      int playerPosY = floor(camera.Position[1] / CY);
6      int playerPosZ = floor(camera.Position[2] / CZ);
7      lastPlayerPos = glm::vec3(playerPosX, playerPosY, playerPosZ);
8
9      // Itera entre os chunks existentes, verificando se estão muito
10     distantes para exclui-los
11     for (ChunkMap::iterator iter = chunks.begin();
12     iter != chunks.end(); ) {
13         if (iter->second->posX / CX > playerPosX + SCX / 2 ||
14         iter->second->posX / CX < playerPosX + (-SCX / 2) ||
15             iter->second->posZ / CZ > playerPosZ + SCZ / 2 ||
16             iter->second->posZ / CZ < playerPosZ + (-SCZ / 2)) {
17             chunks.erase(iter++);
18         }
19         else {
20             ++iter;
21         }
22     }
23
24     // Loop de acordo com a posição da c mera, verificando se existem
25     chunks nestas posições, se não os criando
26     for (int x = playerPosX + (-SCX / 2); x <= playerPosX + SCX / 2; x++)
27         for (int y = 0; y < SCY; y++)
28             for (int z = playerPosZ + (-SCZ / 2);
29             z <= playerPosZ + SCZ / 2; z++) {
30                 key.x = x; key.y = y; key.z = z;
31                 if (!chunks[key]) {
32                     chunks[key] = std::unique_ptr<Chunk>
33                     (new Chunk(x, y, z));
34                 }
35             }
36 }
```

Fonte: Autoria própria (2023).

Listagem 7 – Método "Render" da classe "ChunkManager" atualizado para *chunks* "infinitos"

```
1  void ChunkManager::Render(Shader shader, glm::mat4 &vp) {
2      // Obtêm a posição da c mera e a armazena em variáveis auxiliares
3      int playerPosX = floor(camera.Position[0] / CX);
4      int playerPosY = floor(camera.Position[1] / CY);
5      int playerPosZ = floor(camera.Position[2] / CZ);
6      // Verifica se a posição da c mera mudou, se sim chama o método
7      Update
8      if (playerPosX != lastPlayerPos[0] ||
9      playerPosY != lastPlayerPos[1] ||
10     playerPosZ != lastPlayerPos[2]) {
11         Update();
12     }
13     glm::vec3 pos;
14     if (!chunks.empty()) {
15         for (ChunkMap::iterator iter = chunks.begin();
16         iter != chunks.end(); ++iter) {
17             pos.x = iter->second->posX;
18             pos.y = iter->second->posY; pos.z = iter->second->posZ;
19             glm::mat4 model = glm::translate(glm::mat4(1), pos);
20             shader.setMat4("model", model);
21
22             iter->second->Render();
23         }
24     }
25 }
```

Fonte: Autoria própria (2023).

Listagem 8 – Método auxiliar "GetNeighbourBlock" da classe "Chunk"

```
1  uint8_t Chunk::GetNeighbourBlock(Chunk* neighbour, int x, int y, int z) {
2      if (neighbour)
3          return neighbour->blk[x][y][z];
4      return 0;
5  }
```

Fonte: Autoria própria (2023).

Listagem 9 – Método auxiliar "setNeighbours" da classe "Chunk"

```
1  uint8_t Chunk::GetNeighbourBlock(Chunk* neighbour, int x, int y, int z) {
2      if (neighbour)
3          return neighbour->blk[x][y][z];
4      return 0;
5  }
```

Fonte: Autoria própria (2023).

Listagem 10 – Método "Update" da classe "ChunkManager" atualizado para *face culling*

```
1  void ChunkManager::Update() {
2                                    ...
3  // Itera sobre os chunks existentes
4  for (ChunkMap::iterator iter = chunks.begin(); iter != chunks.end();
5  ++iter) {
6      key.x = iter->second->posX;
7      key.y = iter->second->posY;
8      key.z = iter->second->posZ;
9      // Chama o método "setNeighbours", passando seu chunk vizinho para
10     cada lado caso o mesmo exista, se não passando apenas nullptr
11     iter->second->setNeighbours(chunks.count(glm::ivec3(key.x / CX - 1,
12     key.y / CY, key.z / CZ)) ?
13     chunks[glm::ivec3(key.x / CX - 1, key.y / CY, key.z / CZ)].get() :
14     nullptr, chunks.count(glm::ivec3(key.x / CX + 1, key.y / CY, key.z
15     / CZ)) ?
16     chunks[glm::ivec3(key.x / CX + 1, key.y / CY, key.z / CZ)].get() :
17     nullptr, chunks.count(glm::ivec3(key.x / CX, key.y / CY - 1, key.z
18     / CZ)) ?
19     chunks[glm::ivec3(key.x / CX, key.y / CY - 1, key.z / CZ)].get() :
20     nullptr, chunks.count(glm::ivec3(key.x / CX, key.y / CY + 1, key.z
21     / CZ)) ?
22     chunks[glm::ivec3(key.x / CX, key.y / CY + 1, key.z / CZ)].get() :
23     nullptr, chunks.count(glm::ivec3(key.x / CX, key.y / CY, key.z /
24     CZ - 1)) ?
25     chunks[glm::ivec3(key.x / CX, key.y / CY, key.z /
26     CZ - 1)].get() :
27     nullptr, chunks.count(glm::ivec3(key.x / CX, key.y / CY, key.z /
28     CZ + 1)) ?
29     chunks[glm::ivec3(key.x / CX, key.y / CY, key.z / CZ + 1)].get() :
30     nullptr);
31 }
```

Fonte: Autoria própria (2023).

Listagem 11 – Método "Update" da classe "Chunk" atualizado para *face culling*

```
void Chunk::Update() {
                                        ...
    // LADO X NEGATIVO DO VOXEL
    if ((x == 0 && !GetNeighbourBlock(neighXN, CX - 1, y, z)) ||
       (x > 0 && !blk[x - 1][y][z])) {
        // tri ngulo de baixo
        vertex[i++] = CubeVertex(x, y, z, type, -1, 0, 0);
        vertex[i++] = CubeVertex(x, y, z + 1, type, -1, 0, 0);
        vertex[i++] = CubeVertex(x, y + 1, z, type, -1, 0, 0);
        // tri ngulo de cima
        vertex[i++] = CubeVertex(x, y + 1, z, type, -1, 0, 0);
        vertex[i++] = CubeVertex(x, y, z + 1, type, -1, 0, 0);
        vertex[i++] = CubeVertex(x, y + 1, z + 1, type, -1, 0, 0);
     }
     // LADO X POSITIVO DO VOXEL
                                        ...
```

Fonte: Autoria própria (2023).

Listagem 12 – *Constructor* da classe "ChunkManager" atualizado para geração procedural

```
ChunkManager::ChunkManager() {
    // Geração e definição de um valor aleatório para a seed do
    heightmap
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(std::numeric_limits
    <int>::min(), std::numeric_limits<int>::max());
    myModule.SetSeed(dist(mt));
    // Criação de um heightmap utilizando Perlin Noise, armazenando-o
    no objeto "heightMap" e definindo seu tamanho igual ao número de
    voxels em um chunk
    heightMapBuilder.SetSourceModule(myModule);
    heightMapBuilder.SetDestNoiseMap(heightMap);
    heightMapBuilder.SetDestSize(CX, CZ);

    Update();
}
```

Fonte: Autoria própria (2023).

Listagem 13 – Parte do método "Update" da classe "ChunkManager" adicionado para geração procedural

```
1  // Criação dos chunks em volta da posição da c mera
2      for (int x = playerPosX + (-SCX / 2); x <= playerPosX + SCX / 2; x++)
3          for (int y = 0; y < SCY; y++)
4              for (int z = playerPosZ + (-SCZ / 2); z <= playerPosZ +
5              SCZ / 2; z++) {
6                  key.x = x; key.y = y; key.z = z;
7                  if (!chunks[key]) {
8                      // Definição dos limites do heightmap de acordo
9                      com a posição do chunk, garantindo continuidade dos dados
10                     heightMapBuilder.SetBounds(x / 10.0f,
11                     x / 10.0f + 0.1, z / 10.0f, z / 10.0f + 0.1);
12                     // Geração do heightmap nestes limites
13                     heightMapBuilder.Build();
14                     // Criação do chunk agora passando dados do
15                     heightmap
16                     chunks[key] = std::unique_ptr<Chunk>
17                     (new Chunk(x, y, z, heightMap));
18                 }
19             }
```

Fonte: Autoria própria (2023).

Listagem 14 – *Constructor* da classe "Chunk" atualizado para geração procedural

```
Chunk::Chunk(int i, int j, int k, utils::NoiseMap heightMap) {
    memset(blk, 0, sizeof(blk));
    elements = 0;
    changed = true;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    posX = i * CX;
    posY = j * CY;
    posZ = k * CZ;
    // Definição do valor de altura máximo deste chunk de acordo com
    sua posição global
    float maxHeightValue = (CY * SCY - 1);
    for (int x = 0; x < CX; x++) {
        for (int z = 0; z < CZ; z++) {
            // Obtendo o valor de altura do heightmap de acordo com
            a posição X e Z e posição global do chunk
            float heightmapValue = (heightMap.GetValue(x, z) + 1) / 2;
            float heightValue = heightmapValue * maxHeightValue-posY;
            // Caso o valor de altura seja maior que o valor máximo
            que o chunk pode armazenar, seu valor é alterado para o
            mesmo
            if (heightValue > CY) heightValue = CY;
            // Caso o valor de altura seja menor do que 1, seu valor é
            alterado para 1, garantindo que não existam posições sem
            voxels existentes
            if (posY == 0 && heightValue < 1) heightValue = 1;
            // Criação de voxels até que o valor de elevação seja
            atingido, definindo o tipo do voxel de acordo com sua
            posição de altura global
            for (int y = 0; y < heightValue; y++) {
                if((y + posY) == maxHeightValue)
                    blk[x][y][z] = BlockType_Snow;
                else if ((y + posY) > (maxHeightValue * 0.75))
                    blk[x][y][z] = BlockType_Rock;
                else if ((y + posY) > (maxHeightValue * 0.50))
                    blk[x][y][z] = BlockType_Dirt;
                else if ((y + posY) > (maxHeightValue * 0.15))
                    blk[x][y][z] = BlockType_Grass;
                else if ((y + posY) > (maxHeightValue * 0.05))
                    blk[x][y][z] = BlockType_Sand;
                else
                    blk[x][y][z] = BlockType_Shallow;
            }
        }
    }
}
```

Fonte: Autoria própria (2023).

Listagem 15 – Método "Render" da classe "ChunkManager" atualizado para frustum culling

```
1  void ChunkManager::Render(Shader shader, glm::mat4 &vp) {
2      int playerPosX = floor(camera.Position[0] / CX);
3      int playerPosY = floor(camera.Position[1] / CY);
4      int playerPosZ = floor(camera.Position[2] / CZ);
5      if (playerPosX != lastPlayerPos[0] ||
6      playerPosY != lastPlayerPos[1] ||
7      playerPosZ != lastPlayerPos[2]) {
8          Update();
9      }
10     glm::vec3 pos;
11     if (!chunks.empty()) {
12         for (ChunkMap::iterator iter = chunks.begin();
13         iter != chunks.end(); ++iter) {
14             // Changing each chunk to their position so they aren't
15             stacked on top of eachother
16             pos.x = iter->second->posX;
17             pos.y = iter->second->posY; pos.z = iter->second->posZ;
18             glm::mat4 model = glm::translate(glm::mat4(1), pos);
19             shader.setMat4("model", model);
20
21             // Cálculos para verificar se o chunk está na tela
22             glm::vec4 center = (vp * model) *
23             glm::vec4(CX / 2, CY / 2, CZ / 2, 1);
24             float d = glm::length(center);
25             center.x /= center.w;
26             center.y /= center.w;
27             // Não renderiza este chunk se ele estiver atrás da
28             camera/fora da tela
29             if (center.z < -CY / 2)
30                 continue;
31             if (fabsf(center.x) > 1 + fabsf(CY * 2 / center.w) ||
32             fabsf(center.y) > 1 + fabsf(CY * 2 / center.w))
33                 continue;
34
35             iter->second->Render();
36         }
37     }
38 }
```

Fonte: Autoria própria (2023).

7 RESULTADOS

A maioria das atividades conduzidas geraram resultados promissores. Os aspectos conceituais apresentados no início deste trabalho prepararam a base de conhecimento necessária para o entendimento na geração e utilização de voxels em um aplicativo gráfico. Já o mapeamento sistemático resultou na análise de estudos correlatos, identificando tecnologias e técnicas comumente utilizadas, onde também pode ser observada a falta de estudos que auxiliem na utilização de voxels como representação de terreno. Em sequência, a seção de metodologia detalhou as técnicas encontradas, explicando como funcionam e quais seus prós e contras.

Com estas informações previamente levantadas, um algoritmo que faz a geração procedural "infinita" de terreno com voxels também foi desenvolvido utilizando C++ e OpenGL, além das técnicas de *geometry batching*, *chunking*, armazenamento de dados com *hashmap*, *face culling*, *frustum culling* e geração procedural. O algoritmo foi testado em uma máquina utilizando o sistema operacional Windows 11, com as seguintes especificações: Processador Ryzen 5600X; Placa de vídeo GeForce RTX 2070 Super e Memória RAM 32GB.

Com o algoritmo finalizado, seu desenvolvimento foi detalhado neste trabalho em etapas de forma incremental, a fim de prover ao leitor um maior entendimento do processo de desenvolvimento e estrutura do algoritmo com diagramas UML, além de conhecimento prático das técnicas analisadas e escolhidas.

Como resultado, o algoritmo atingiu o objetivo de realizar a geração procedural "infinita" de terreno com voxels, com código realizado de forma simples e objetiva, além de amplamente comentado para facilitar seu entendimento. Sua estrutura final está disponibilizada abaixo na Figura 23 com um diagrama UML, além de imagens da aplicação em execução nas Figuras 24 e 25 em sequência.

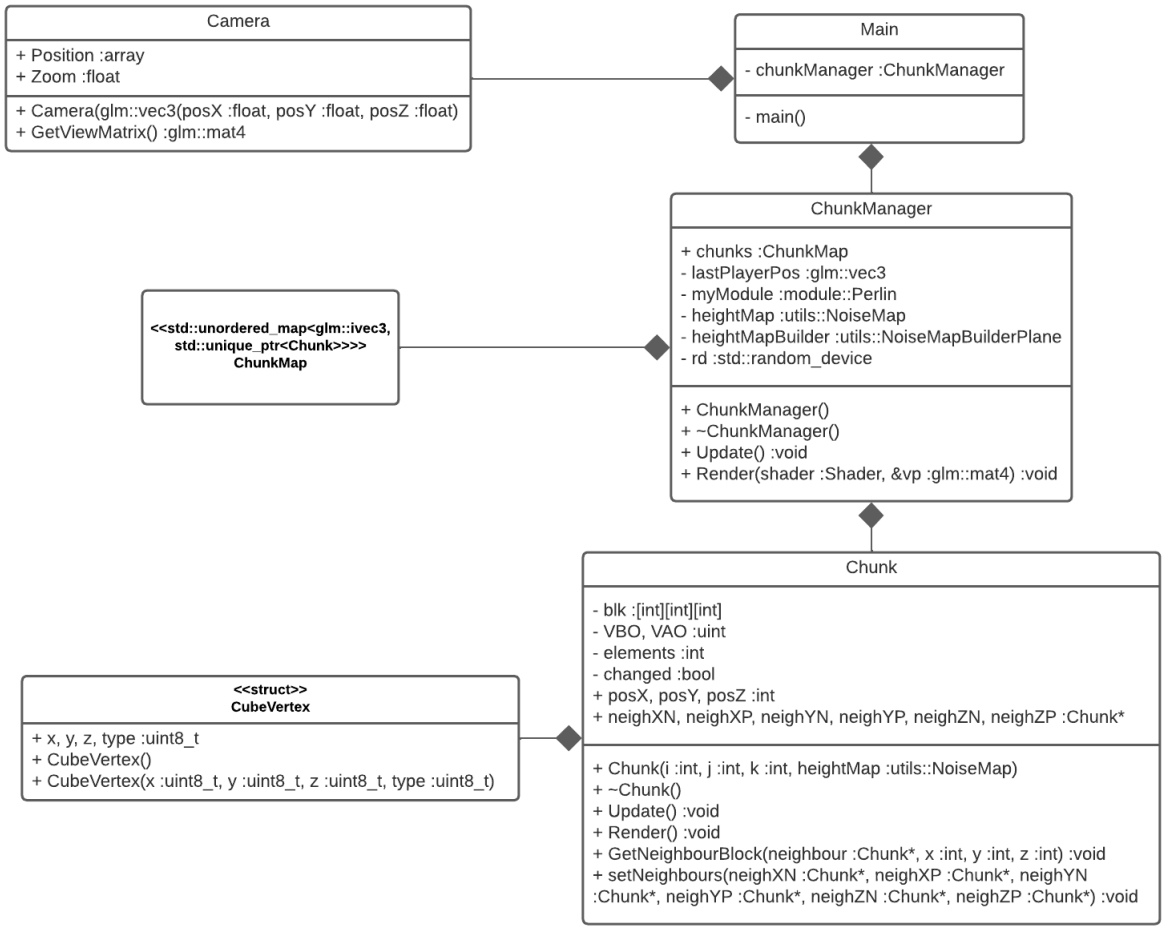
Conforme descrito na seção 5.3, inicialmente o desenvolvimento do algoritmo também englobava a aplicação das técnicas de *ambient occlusion* e *multithreading*, porém, ambas as técnicas apresentaram problemas em suas aplicações.

A técnica de *ambient occlusion* pode ser considerada complexa quando não se há muita experiência com *shaders* e OpenGL, o que resultou em uma aplicação não satisfatória com problemas, sendo descartada pelo tempo limitado de implementação para este trabalho e por ter sua necessidade reavaliada, pois após a fácil implementação de *ambient* e *diffuse lighting* que são de algoritmos básicos de iluminação, o terreno obteve bons resultados de visibilidade.

Já a técnica de *multithreading* se apresentou complexa por necessitar de uma grande reestruturação do código, já que na estrutura apresentada a modificação e renderização de *chunks* estão interligadas fortemente a fim de simplificar o algoritmo, o que resultou novamente em seu descarte pelo tempo limitado para sua implementação.

Por fim, este trabalho como um todo apresenta resultados satisfatórios em comparação aos identificados na seção de mapeamento sistemático, pois os poucos estudos com mesma

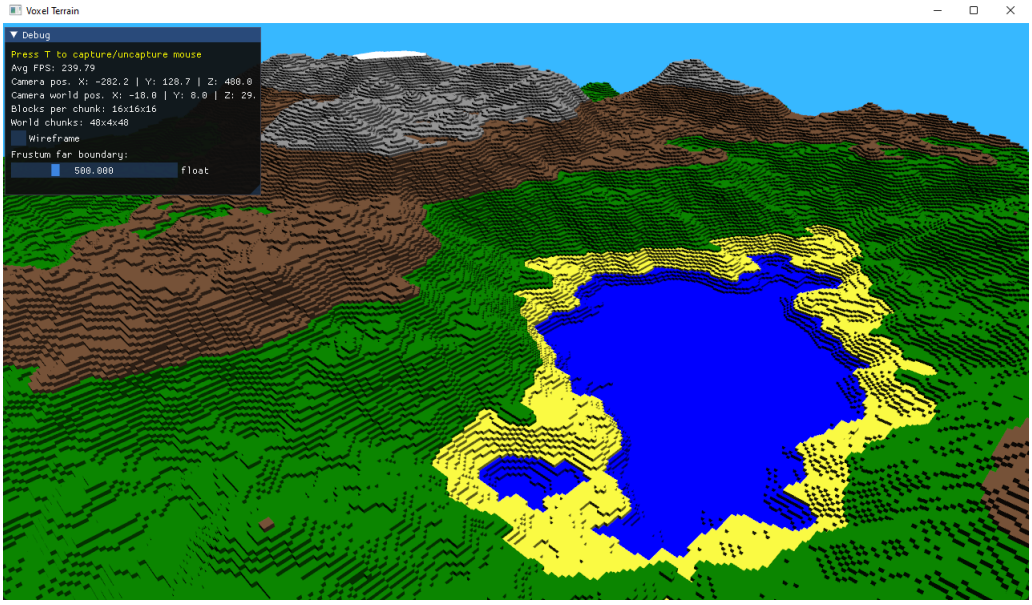
Figura 23 – Diagrama UML final do algoritmo



Fonte: Autoria própria (2023).

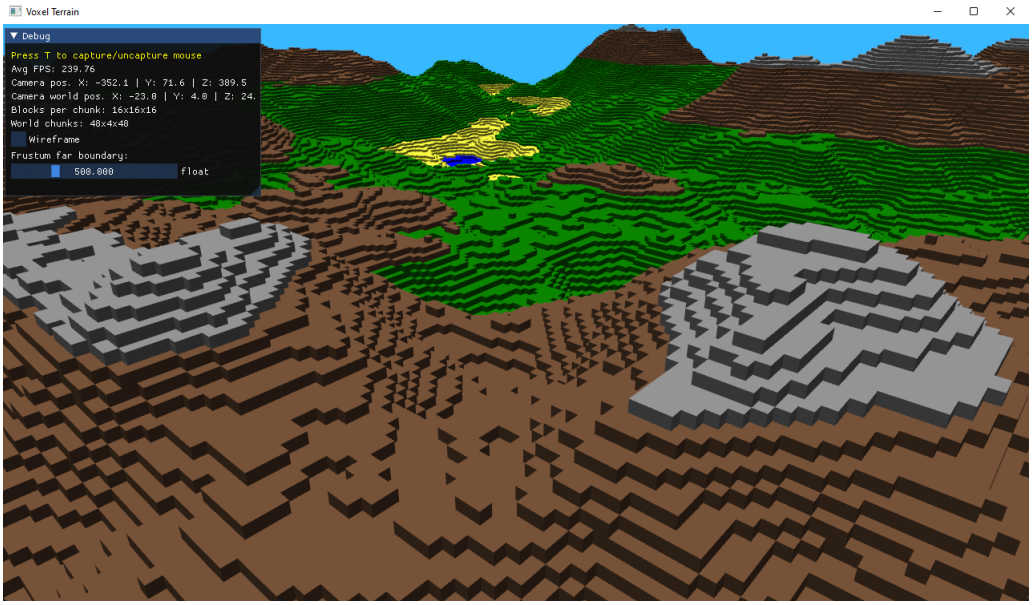
problemática não apresentaram embasamento teórico da área e estudo das técnicas existentes, além de falharem em prover documentação do código desenvolvido.

Figura 24 – Resultado final do algoritmo - 1



Fonte: Autoria própria (2023).

Figura 25 – Resultado final do algoritmo - 2



Fonte: Autoria própria (2023).

8 CONCLUSÃO

Existe uma lacuna de pesquisa (*gap*) considerável em relação à documentação das pesquisas relacionadas à geração de terrenos procedurais com voxels e feitos em C++ e OpenGL. A fim de contribuir para solucionar esta lacuna, este trabalho apresentou a proposta de desenvolvimento e documentação de todas as etapas da construção de um algoritmo de geração de terrenos procedurais com voxels e programado C++ e OpenGL. Por meio do mapeamento sistemático realizado, foi possível confirmar a deficiência na documentação sistematizada desta área de pesquisa e também compreender mais precisamente o estado da arte atual. Os resultados do mapeamento contribuíram para documentar a falta de recursos acadêmicos que tratem do assunto com o detalhamento e a sistematização esperadas em trabalhos científicos. A metodologia detalhou técnicas comumente utilizadas, e o desenvolvimento documentou o desenvolvimento do algoritmo de forma incremental. Pretende-se que o conhecimento adquirido ao final deste trabalho possa contribuir para sanar essa lacuna de pesquisa na área, auxiliando pesquisadores e desenvolvedores interessados em utilizar voxels como forma de representação de volume, principalmente na área de desenvolvimento de jogos.

8.0.1 TRABALHOS FUTUROS

Como trabalhos futuros, pode-se realizar a implementação de um algoritmo de mesmo fim utilizando-se de outras tecnologias ou técnicas. Abaixo são descritas, de modo não exaustivo, algumas possibilidade:

- Vulcan, por ser uma API de renderização mantida pelo mesmo grupo do OpenGL, porém, com objetivo de ser uma API para a nova geração de gráficos (KHRONOS, 2022);
- Rust, por ser uma linguagem com bom desempenho e gerenciamento de memória automático (TEAM, 2022);
- Octrees, pois suas subdivisões de espaço encaixam muito bem para a separação de um terreno com voxels, além de terem uma implementação muito diferente das demais técnicas de armazenamento, conforme explicado na Seção 5.2.2.2.

Ademais, trabalhos futuros podem também envolver a implementação de técnicas não utilizadas neste trabalho, como *multithreading*, *ambient occlusion* e *greedy meshing*.

REFERÊNCIAS

BRUMMELEN, J. V.; CHEN, B. **. Procedural Generation**. 2018. Disponível em: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html. Acesso em: 03 de out. de 2021.

CBOVIK, A. C. **Handbook of Image and Video Processing**. 2ed. ed. [*S.l.*]: MA: Elsevier Academic Press, 2005.

CHEN, M.; KAUFMAN ARIE E. ANDYAGEL, R. **Volume Graphics**. 1 ed. ed. [*S.l.*]: Springer-Verlag London, 2000.

CHIEN-WEN, C. *et al.* Sa real-time sculpting and terrain generation system for interactive content creation. *In*: **IEEE Access**. [*S.l.*: *s.n.*], 2021.

CNT. **Custo logístico consome 12,7% do PIB do Brasil**. 2003. Disponível em: https://libnoise.sourceforge.net/. Acesso em: 5 de novembro de 2022.

DIEBOLD, P.; VETRO, A.; FERNANDEZ, D. M. An exploratory study on technology transfer in software engineering. *In*: **2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [*S.l.*: *s.n.*], 2015. p. 1–10.

DORAN, J.; PARBERRY, I. Controlled procedural terrain generation using software agents. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 2, n. 2, p. 111–119, 2010.

DUSTERWALD, S. **Procedural Generation of Voxel Worlds with Castles**. 2015. Tese (Dissertação em Computação) — The University of Waikato, Hamilton, 2015.

FOLEY, J. D. e. a. **Introduction to Computer Graphics**. 1 st. ed. [*S.l.*]: Addison Wesley, 1993.

GAO, K.; HE, J.; QI, Y. A relevant research on the establishment of a voxel gaming world. *In*: **2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)**. [*S.l.*: *s.n.*], 2018. p. 1–2.

KHRONOS, G. **OpenGL Overview**. 2022. Disponível em: https://www.khronos.org/opengl/. Acesso em: 20 de mai. de 2022.

KIRSI, F.-E. **Voxel Game Engine Using Blender Game Engine**. 2016. Tese (Dissertação em Computação) — Tallinn University of Technology, Hamilton, 2016.

LYSENKO, M. **An Analysis of Minecraft-like Engines**. 2012. Disponível em: https://0fps.net/2012/01/14/an-analysis-of-minecraft-like-engines. Acesso em: 04 de out. de 2021.

LYSENKO, M. **Meshing in a Minecraft Game**. 2012. Disponível em: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. Acesso em: 04 de out. de 2021.

LYSENKO, M. **Smooth Voxel Terrain**. 2012. Disponível em: https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/. Acesso em: 04 de out. de 2021.

LYSENKO, M. **Ambient occlusion for Minecraft-like worlds**. 2013. Disponível em: https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/. Acesso em: 23 de mai. de 2022.

OGAYAR, C.; RUEDA, A.; SEGURA, R. Fast and simple hardware accelerated voxelizations using simplicial coverings. **Visual Comput**, v. 23, p. 534–543, 2007.

OLIVEIRA, C.; OLIVEIRA, F.; PEDRINI, H. **Game Engine With 3D Graphics**. 2016. Tese (Projeto de Graduação) — Instituto de Computação - Universidade Estadual de Campinas, Campinas, 2016.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and Software Technology**, v. 64, p. 1–18, 2015.

SANTAMARÍA-IBIRIKA, A. *et al.* Procedural approach to volumetric terrain generation. Springer-Verlag, v. 30, n. 9, p. 997–1007, 2014.

SHORT, T.; ADAMS, T. **Procedural Generation in Game Design**. 1 st. ed. [*S.l.*]: A K Peters/CRC Press, 2017.

TEAM, R. **Rust Programming Language**. 2022. Disponível em: https://www.rust-lang.org/. Acesso em: 14 de nov. de 2022.

TRAPP, M.; DÖLLNER, J. Geometry batching using texture-arrays. *In*: **0th International Conference on Computer Graphics Theory and Applications (GRAPP)**. [*S.l.*: *s.n.*], 2015. p. 1–9.

VRIES, J. **Learn OpenGL: Learn OpenGL Graphics programming in a step-by-step fashion**. 1 st. ed. [*S.l.*]: Kendall Welling, 2020.

WILDER, M. **An Investigation In Implementing a C++ Voxel Game Engine with Destructible Terrain**. 2015. Tese (Projeto de Pesquisa) — University of Akron, Akron, 2015.

WILLIAMS, A.; WILLIAMS, A. **C++ Concurrency in Action: Practical Multithreading**. 1 st. ed. [*S.l.*]: Manning Publications, 2012.