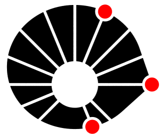

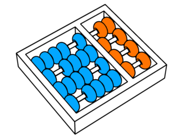

Uso de Sparse Voxel Octrees para Ray Tracing em Tempo Real

Luiz Henrique Aguiar de Lima Alves *Hélio Pedrini*
José Mario De Martino

Relatório Técnico - IC-PFG-24-18
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Uso de Sparse Voxel Octrees para Ray Tracing em Tempo Real

Luiz Henrique Aguiar de Lima Alves* Hélio Pedrini*

José Mario De Martino†

Resumo

Este trabalho explora a aplicação do paradigma de renderização por *Ray Tracing* para a visualização em tempo real de modelos descritos por voxels. Descrevemos inicialmente uma implementação usando *Compute Shaders* com os dados organizados em uma matriz tridimensional (3D). Em seguida, introduzimos uma otimização trocando a estrutura de dados para uma *Sparse Voxel Octree* (SVO), explicando sua construção, como pode ser percorrida, bem como os ganhos de eficiência em termos de memória e processamento gráfico alcançados com ela. Finalmente, demonstra-se a importância dessa estrutura de aceleração para alcançar métricas satisfatórias de desempenho em aplicações interativas.

1 Introdução

No início dos anos 2010, o jogo *Minecraft* [9] popularizou o uso de voxels – pequenos blocos tridimensionais que formam estruturas visuais complexas - como elemento gráfico e mecânico. A possibilidade de representar o mundo do jogo volumetricamente, permitindo que os jogadores manipulem o ambiente de forma intuitiva e criativa, transformou o jogo em um fenômeno cultural. Essa abordagem inovadora não apenas capturou a imaginação de milhões de jogadores, mas também inspirou uma nova geração de jogos e aplicações interativas que exploram a representação voxelizada de ambientes virtuais.

Com os avanços gráficos recentes, a renderização de voxels evoluiu significativamente. A crescente capacidade computacional de hardware moderno e o desenvolvimento de algoritmos que utilizem esses recursos abriram caminho para a utilização de voxels em maior granularidade e com maior realismo visual. Exemplos notáveis dessa evolução incluem o jogo *Teardown* [13] e as demonstrações do motor de voxels apresentado por Lin [8], que alcançam níveis de detalhamento e interatividade que antes eram considerados inviáveis.

Nesse contexto, observa-se, nos últimos anos, o início de uma transformação significativa na maneira como jogos e outras aplicações interativas utilizam voxels. Este trabalho busca então explorar algumas das técnicas que servem como base para essa nova forma de uso de voxels.

*Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852

†Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852

2 Conceitos Teóricos

Antes de abordar os detalhes do trabalho, é fundamental apresentar e explicar os conceitos teóricos necessários para sua compreensão.

2.1 Voxel

O termo "voxel" se refere a uma unidade volumétrica tridimensional que representa um ponto em um espaço discretizado. Essa unidade é análoga ao pixel, que representa uma unidade bidimensional em imagens.

Os voxels são comumente utilizados em representações espaciais discretas, como em jogos, simulações e aplicações médicas (como imagens de tomografia computadorizada), onde o ambiente 3D é decomposto em pequenos blocos que facilitam a manipulação e visualização de grandes volumes de dados.

2.2 Ray Tracing

O *ray tracing* [5, 11, 12] é uma técnica de renderização que simula o comportamento da luz ao traçar o caminho de raios em uma cena tridimensional partindo de uma câmera. Essa abordagem permite calcular efeitos visuais realistas, como sombras, reflexos e refrações.

Existem diferentes formas de implementação do *ray tracing*, dependendo do hardware e das necessidades de desempenho:

- **Na CPU:** Oferecem maior flexibilidade e precisão, mas têm desempenho limitado devido à menor capacidade de paralelismo. Isso as torna inadequadas para aplicações em tempo real.

- **Na GPU com *compute shaders*:** Aproveitam o paralelismo massivo das GPUs modernas, possibilitando renderizações em tempo real com boa eficiência, entretanto, é necessário implementar grande parte da lógica manualmente. Esta abordagem é adotada neste projeto devido à combinação de flexibilidade e alto desempenho.

- **Na GPU com hardware dedicado:** GPUs modernas possuem núcleos especializados para *ray tracing*, como os *RT cores*, que aceleram a técnica significativamente. Essa solução é ideal para renderizações de alta qualidade em tempo real, mas está restrita a dispositivos que possuem esse tipo de hardware.

2.3 Sparse Voxel Octrees

As *Sparse Voxel Octrees* (SVO) [7] são uma estrutura de dados hierárquica que permite representar e organizar de forma eficiente volumes compostos por voxels. Essa técnica se baseia na subdivisão recursiva do espaço 3D em oito octantes menores, criando uma árvore de subdivisões onde cada nó representa uma região do espaço, sendo ilustrada na Figura 1.

Ao contrário das *octrees* tradicionais, as SVOs economizam memória ao representar apenas as áreas de interesse, ignorando regiões vazias ou homogêneas. Essa compactação permite acessar voxels de forma eficiente e reduz o tempo de processamento, especialmente

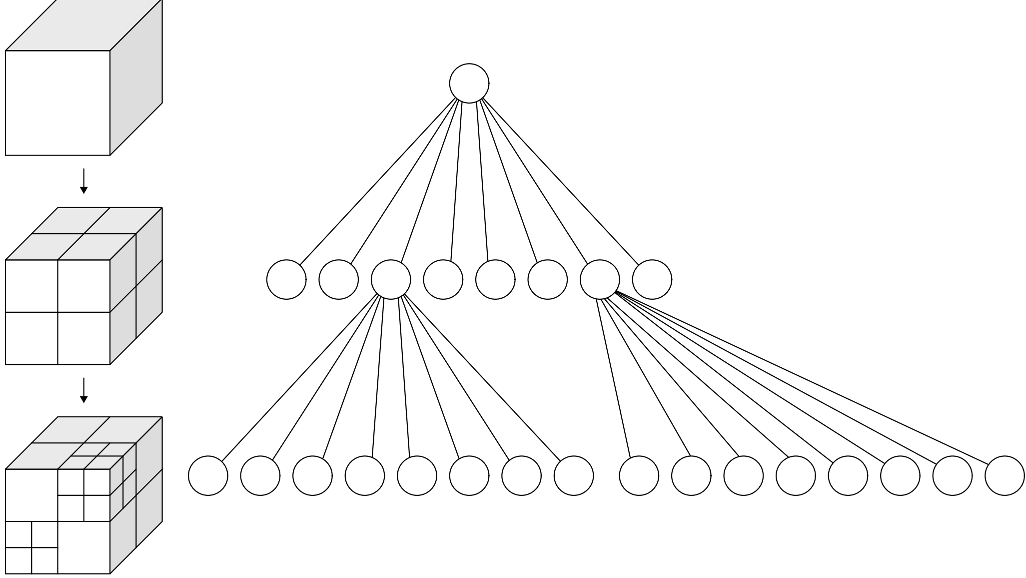

Figura 1: Representação de uma SVO. Imagem extraída de [15].

em ambientes de *ray tracing*, onde a hierarquia da árvore permite saltar diretamente para regiões de relevância, otimizando a trajetória dos raios e, consequentemente, o tempo de renderização.

3 Objetivos

Este trabalho tem como objetivo principal explorar o uso do *ray tracing* para a renderização de estruturas voxelizadas em aplicações interativas, demonstrando sua viabilidade para uso em tempo real.

A proposta é desenvolver um *ray tracer* de voxels como prova de conceito, implementando uma otimização baseada em *Sparse Voxel Octrees* (SVOs) como estrutura de aceleração. Ao demonstrar a viabilidade do *ray tracing* em ambientes voxelizados, o trabalho visa também avaliar o impacto dessa otimização no desempenho, analisando o ganho de eficiência alcançado.

4 Metodologia

O projeto foi desenvolvido na linguagem de programação `C++` utilizando a API gráfica `Vulkan` e seu código encontra-se disponível no GitHub: `https://github.com/HikkusT/hik-voxel`. Essas tecnologias foram escolhidas por permitirem um controle fino sobre os recursos do sistema, já que essa aplicação conta com uma sensibilidade alta em relação ao desempenho. Essa combinação possibilita, por exemplo, o controle explícito sobre o layout de memória e a transferência de dados para a GPU, ambos essenciais para otimizar o processamento gráfico.

Além disso, tratando-se de uma aplicação gráfica, é fundamental definirmos os elementos a serem renderizados. Como o projeto é especificamente relacionado a voxels, é necessário que esses elementos tenham uma representação voxelizada. Para atender a essa necessidade, o projeto utiliza modelos no formato `.vox` [4], um formato especificado pelos desenvolvedores do *MagicaVoxel* que tornou-se um padrão de referência nesse tipo de representação.

Apesar disso, a disponibilidade de modelos em formato `.vox` na web ainda é relativamente limitada. Para ampliar a gama de opções, este projeto utiliza a ferramenta *Voxelizer* [14], que converte modelos de outros formatos populares, como `.obj` e `.fbx` [2], para o formato `.vox`, aumentando significativamente o conjunto de modelos acessíveis. A leitura e manipulação desses arquivos `.vox` foram realizadas por meio da biblioteca *OpenGameTools* [10], que fornece uma API em `C++`.

O processo de renderização foi implementado utilizando *Compute Shaders* para realizar os cálculos diretamente na GPU. O resultado do *ray tracing* é gravado em uma textura, que posteriormente é copiada para o *framebuffer*, permitindo a exibição final.

Com essa infraestrutura estabelecida, foram desenvolvidas duas abordagens principais para o *ray tracing* de voxels: uma implementação ingênua baseada em matriz tridimensional e uma versão otimizada utilizando *Sparse Voxel Octrees* (SVO). Ambas as abordagens são detalhadas a seguir.

4.1 Implementação Ingênua

A abordagem inicial para o *ray tracing* de voxels é utilizar uma matriz tridimensional para representar o espaço discreto. Isso é expresso na equação $F(x, y, z) = \texttt{world}[\lfloor x \rfloor][\lfloor y \rfloor][\lfloor z \rfloor]$, onde $F(x, y, z)$ é o valor do voxel na posição (x, y, z) do mundo, e `world` é a matriz tridimensional usada como estrutura de dados.

Para enviar essa matriz para a GPU, ela é linearizada em um único vetor unidimensional, utilizando a fórmula $\texttt{world}[x][y][z] \rightarrow \texttt{data}[x+\texttt{size}\cdot y+\texttt{size}^2\cdot z]$. Esses dados são declarados no *shader* assim como mostrado no Código 1.

```
layout(set = 0, binding = 1) buffer World {
    float voxelSize;
    int worldSize;
    int data[];
} world;
```

Código 1: Estrutura de dados serializada na GPU na implementação ingênua.

O *ray tracing* nesta abordagem foi implementado com base no algoritmo descrito por Amanatides et al. [1]. Esse método, uma variação de *Digital Differential Analyzer* (DDA), itera eficientemente voxel a voxel no caminho de um raio até a colisão com um voxel sólido ou escapar do volume delimitado.

A Figura 2 apresenta uma visualização de depuração que ilustra o funcionamento do algoritmo. Nessa figura, o nível de branco indica o número de iterações realizadas pelo raio em cada pixel; áreas mais brancas representam um maior número de iterações.

Figura 2: Visualização de *debug* de *ray tracing* usando grade uniforme. O nível de branco representa quantidade de iterações.

4.2 Otimização com SVO

Visando melhorar a eficiência do algoritmo baseado em matrizes tridimensionais, foi implementada uma otimização utilizando *Sparse Voxel Octrees* (SVO), conforme descrito por Laine e Karras [6]. Nessa estrutura, o espaço tridimensional é representado por uma árvore onde cada nó não-folha possui oito filhos correspondentes às subdivisões do espaço que ele representa. Os nós folha armazenam um único valor, indicando que todos os voxels da região correspondente possuem esse mesmo valor.

Para transferir a estrutura da SVO para a GPU, é necessário serializá-la por meio de um processo de linearização. Esse processo percorre a árvore em profundidade, transformando a hierarquia em uma representação linear descrita no Código 2, onde cada elemento contém:

- **LeafMask:** Uma máscara de bits indicando quais filhos são folhas.
- **childrenOffsets:** Um vetor de inteiros contendo informação sobre os filhos do nó. Para os filhos que são nós-folhas, o valor daquele nó é armazenado. Para os restantes, os índices no vetor que têm o elemento daquele nó são armazenados.

A Figura 3 ilustra diretamente a relação da SVO com esse vetor. Como a árvore é percorrida em profundidade, é sempre possível saber qual o próximo índice disponível no vetor para encaixar o próximo nó.

Com a SVO carregada na GPU, o *ray tracing* foi adaptado para lidar com voxels de tamanhos não uniformes. Para isso, o algoritmo de Amanatides et al. [1] foi substituído por um método que realiza uma série de interseções entre o raio e as AABBs (*Axis-Aligned Bounding Boxes*) dos nós da árvore. Cada interseção com um nó não sólido determina o avanço do algoritmo até a próxima interseção, utilizando o ponto de saída da interseção

```
struct SvoNode {
    int LeafMask;
    int childrenOffsets[8];
};

layout(set = 0, binding = 1) buffer World {
    float voxelSize;
    int worldSize;
    SvoNode data[];
} world;
```

Código 2: Descrição da SVO serializada.

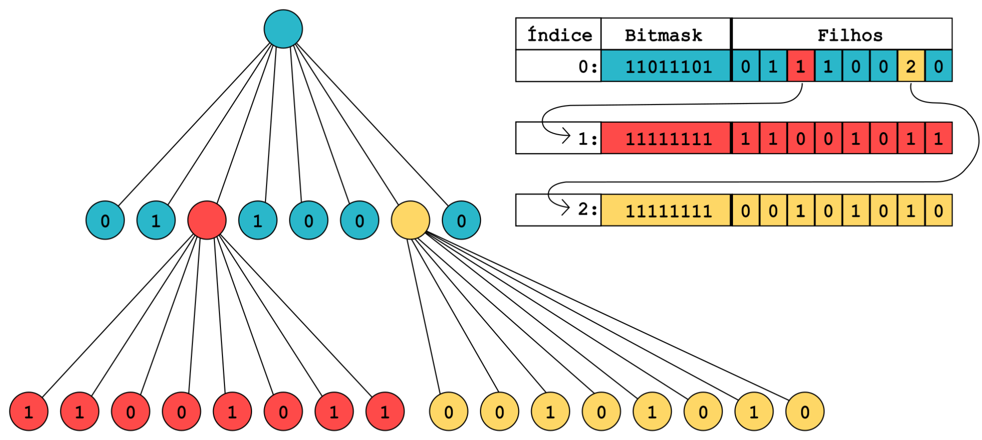

Figura 3: Ilustração do processo de serialização de uma SVO. O nó de índice 0, representado por azul, tem dois filhos não folhas, conforme preenchido na sua máscara de bits.

anterior como início do próximo cálculo. O cálculo das interseções é realizado utilizando o algoritmo descrito por Williams et al. [16], que otimiza a quantidade de operações ponto-flutuante.

Para determinar a AABB correspondente a um voxel em uma dada posição, o algoritmo percorre a árvore, descendo a partir da raiz até chegar no nó folha correspondente. A profundidade da árvore indica o tamanho da AABB, enquanto o caminho percorrido pelos filhos da árvore determina a posição espacial da AABB. A Figura 4 apresenta uma visualização de depuração em que as AABBs dos nós da SVO são destacadas em preto e branco, permitindo observar a granularidade espacial da estrutura.

Outra forma de analisar o comportamento da otimização é observar a Figura 5, que

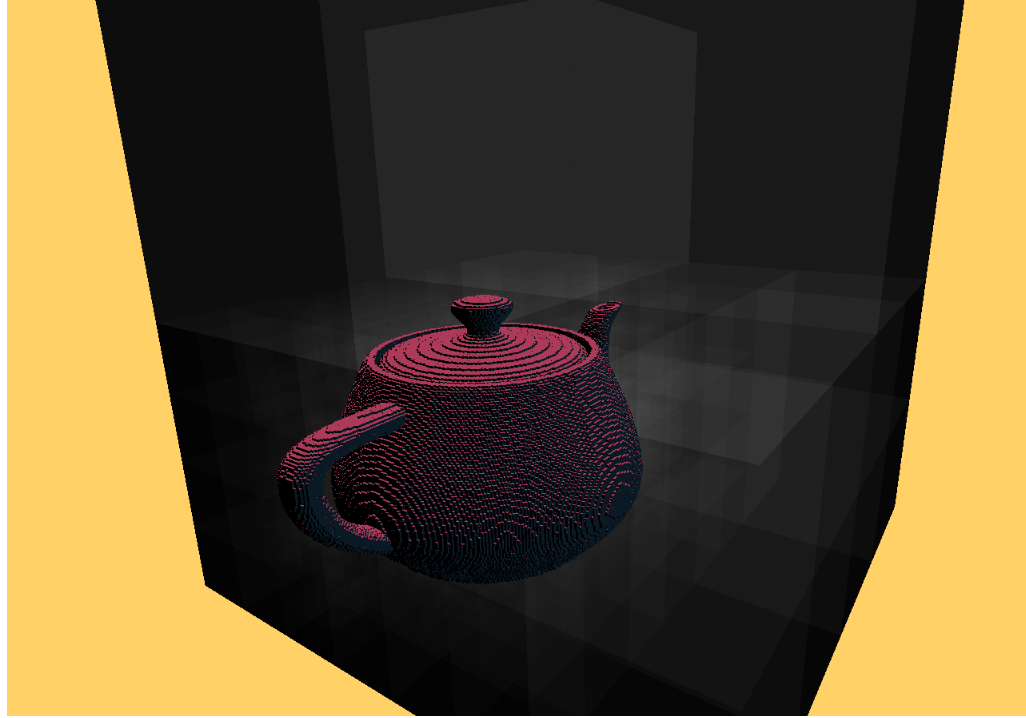

Figura 4: Visualização de modelo em modo depuração da SVO.

apresenta uma visualização equivalente à Figura 2, porém com o brilho aumentado artificialmente para facilitar a observação dos detalhes. É possível notar o "contorno" formado pelas partes não uniformes da grade, evidenciando a granularidade adaptativa característica da SVO.

5 Resultados

Com o objetivo de avaliar a viabilidade da implementação proposta, foram selecionados três modelos 3D, resumidos na Tabela 1. Cada modelo representa um nível diferente de complexidade geométrica, permitindo uma análise abrangente do desempenho do sistema em diferentes cenários.

O *toroide* é uma forma geométrica simples, com baixa resolução de voxels, representando um caso de baixa complexidade. O *bule de chá*, conhecido na literatura como *Utah Teapot* e frequentemente utilizado como *benchmark* em gráficos computacionais, foi escolhido como um caso de complexidade intermediária. Por fim, o modelo da *Pietà*, baseado na escultura de Michelangelo, representa um cenário de alta complexidade geométrica devido à riqueza de detalhes e ao elevado número de voxels.

Para avaliar o desempenho, foram consideradas três métricas principais: o uso de memória da estrutura de dados, o tempo de execução do *ray tracer* e o tempo de geração da estrutura de dados. Essas métricas foram coletadas utilizando o *RenderDoc* [3], uma

Figura 5: Visualização de *debug* de *ray tracing* usando SVO. O nível de branco representa quantidade de iterações.

Tabela 1: Informação dos modelos usados.

Nome	Toroide	Bule	Pietà
Visualização			
Resolução	$64 \times 18 \times 64$	$158 \times 125 \times 255$	$510 \times 484 \times 333$
Voxels Sólidos	$41,032$	$101,683$	$10,357,736$

ferramenta amplamente empregada para análise de aplicações gráficas, capaz de fornecer informações detalhadas sobre a execução de programas na GPU.

Os experimentos foram realizados em um ambiente de execução equipado com uma GPU NVIDIA GeForce RTX 3080 com 12 GB de memória VRAM e com uma resolução de 1700×900 pixels. Os resultados obtidos para cada modelo estão apresentados na Tabela 2.

Os resultados demonstram claramente a eficiência proporcionada pela adoção da *Sparse Voxel Octree* como estrutura de aceleração para o *ray tracing*, tanto em termos de memória quanto de tempo de processamento.

No caso do modelo mais complexo, a Pietà, a abordagem baseada em matriz tridimensional requer 512 MB de memória, enquanto a SVO reduz este consumo para apenas 12,12 MB, uma economia de aproximadamente **97,63%**. Além disso, o tempo de renderização caiu de 150,497 ms para 3,702 ms, resultando em um *speedup* de aproximadamente **40,66 vezes**, viabilizando a execução em tempo real.

Tabela 2: Resultados de desempenho de cada modelo.

Métrica		**Toroide**	**Bule**	**Pietà**
Uso de memória (MB)	Matriz	1.0 MB	64.0 MB	512.0 MB
	SVO	0.07 MB	1.26 MB	12.12 MB
Tempo de renderização (ms)	Matriz	1.795 ms	10.774 ms	150.497 ms
	SVO	1.214 ms	4.925 ms	3.702 ms
Tempo de carregamento (s)	Matriz	$< 1ms$	0.009 s	0.075 s
	SVO	0.067 s	4.121 s	34.636 s

Mesmo em modelos menos complexos, como o toroide, a economia de memória continua relevante, alcançando uma redução de **93%**. Embora o impacto no tempo de renderização seja menos expressivo em modelos simples, a abordagem com SVO ainda apresenta uma leve vantagem em termos de desempenho computacional com um *speedup* de aproximadamente **1,48 vezes**.

Embora a utilização da SVO traga benefícios significativos em tempo de execução, é importante destacar que a sua geração inicial possui um custo elevado. Para cenas complexas, como a Pietà, o tempo de construção da SVO foi de aproximadamente 35 segundos. Mesmo considerando que a implementação atual não está otimizada, ainda é um número grande demais para ser aceitável. Esses resultados evidenciam a importância de pré-computar a SVO sempre que possível, minimizando custos de carregamento. Além disso, vale ressaltar que modificações pontuais em uma SVO já construída são consideravelmente mais eficientes do que reconstruí-la completamente.

6 Conclusões e Trabalhos Futuros

Os resultados apresentados neste trabalho destacam a importância de estruturas de aceleração, como a *Sparse Voxel Octree* (SVO), para viabilizar o uso de *ray tracing* em tempo real em cenas voxelizadas complexas.

Apesar de a técnica de SVO ter sido introduzida há cerca de uma década no trabalho de Laine e Karras [6], ela permanece como uma abordagem de ponta com uso limitado na indústria, possivelmente devido aos poucos recursos disponíveis sobre o tópico. Este projeto contribui para o avanço da área ao servir como uma referência prática e abrangente, abordando o fluxo completo da implementação – desde a estrutura da pipeline gráfica até a execução do *ray tracing* – em vez de focar em aspectos isolados.

Diversas oportunidades de melhoria foram identificadas ao longo do desenvolvimento. A principal delas é a otimização do processo de construção da SVO, que atualmente é extremamente custoso em termos de tempo de processamento, limitando o carregamento a SVO's

pre-computadas. Outra melhoria relevante está no código de *ray tracing*, que apresenta múltiplas ramificações prejudiciais ao desempenho da GPU. Investigar o uso de estruturas alternativas, como uma pilha para a travessia da SVO, pode mitigar esse problema e explorar de forma mais eficiente as características arquiteturais das GPUs modernas.

Por fim, este trabalho deve ser considerado como uma prova de conceito, com vários outros assuntos passíveis de serem estudados. Entre as possíveis extensões estão: suporte a múltiplos materiais, integração de modelos de iluminação mais sofisticados, representação de múltiplos objetos na cena e modificação interativa dos voxels. Essas funcionalidades são necessárias para tornar esse projeto em uma aplicação realmente robusta.

Referências

[1] J. Amanatides, A. Woo, et al. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics*, volume 87, pages 3–10. Citeseer, 1987. 4, 5

[2] Autodesk Inc. *FBX File Format Overview*. `https://www.autodesk.com/products/fbx/overview`. 4

[3] Crytek. RenderDoc Documentation, 2024. `https://renderdoc.org/`. 7

[4] Ephtracy. *MagicaVoxel File Format (.vox)*, 2018. `https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt`. 4

[5] A. S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989. 2

[6] S. Laine and T. Karras. Efficient Sparse Voxel Octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, page 55–63, New York, NY, USA, 2010. Association for Computing Machinery. 5, 9

[7] S. Laine and T. Karras. Efficient Sparse Voxel Octrees. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 55–63, 2010. 2

[8] J. Lin. New Voxel Engine Reveal - Crystal Islands Experiment, 2021. `https://www.youtube.com/watch?v=8ptH79R53c0`. 1

[9] Mojang Studios. Minecraft, 2011. `https://www.minecraft.net/`. 1

[10] J. Paver. Open Game Tools. `https://github.com/jpaver/opengametools`. 4

[11] P. Shirley. Ray Tracing in One Weekend. *Amazon Digital Services LLC*, 1:4, 2018. 2

[12] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. AK Peters, Ltd., 2008. 2

[13] Tuxedo Labs. Teardown, 2020. `https://www.teardowngame.com/`. 1

[14] A. Westerdiep. Voxelizer. `https://drububu.com/miscellaneous/voxelizer/?out=vox`. 4

[15] Wikimedia Commons. File:Octree2.svg — Wikimedia Commons, the free media repository, 2020. `https://commons.wikimedia.org/w/index.php?title=File:Octree2.svg&oldid=484954571`. 3

[16] A. Williams, S. Barrus, R. K. Morley, and P. Shirley. An Efficient and Robust Ray-box Intersection Algorithm. In *ACM SIGGRAPH 2005 Courses*, pages 9–15. ACM, 2005. 6