

A Lightweight, Swappable Voxel Ray-Tracer

Ann Gao
Dept. Computer Science
University of Central Florida
Orlando, FL
an863834@ucf.edu

Nathan Hall
Dept. Computer Science
University of Central Florida
Orlando, FL
na116376@ucf.edu

Jonah Henriksson
Dept. Computer Science
University of Central Florida
Orlando, FL
jo295297@ucf.edu

Christopher Jenkins
Dept. Computer Science
University of Central Florida
Orlando, FL
ch119158@ucf.edu

Felipe Schmidt
Dept. Computer Science
University of Central Florida
Orlando, FL
fe805154@ucf.edu

Abstract—Voxel rendering is a technique for visualizing 3D volumes, with applications in medical technology, terrain visualization, and video games. Simple voxel rendering implementations rely on “meshing”, a process by which the volume is converted to a mesh for rasterization. For many tasks, this is sufficient, but also suffers from memory constraints which prevents meshing from scaling to large scenes (since both the voxel data and heavy mesh representation has to be stored). An alternative solution that lacks such constraints is ray-tracing; which operates on the voxel data itself. In this project, we present a CPU voxel ray-tracer implemented in Rust, alongside two storage solutions that can be swapped: a dense array-backed storage, and a sparse octree storage. The relative performance of these solutions are assessed via render benchmarks.

Index Terms—voxels, ray-tracing, octree.

I. INTRODUCTION

The term “voxels” is derived from the term “pixels”: where a “pixel” is a “picture element” that represents part of a picture, a “voxel” is a “volume element” that represents part of a volume. Volumetric data can range from MRI slices to point clouds to generated volumes in video games. Voxels are a representation of this volumetric data in a grid structure, just as pixels store raster images in a grid structure. However, while pixels and voxels are similar in structure, the way they are visualized differs vastly. Pixels can simply be drawn to a screen surface, but voxels are points in 3D space and drawing techniques can either draw the individual points, as with point cloud rendering, or the isosurface, which is the concern of this paper.

Isosurface rendering is the ubiquitous form of voxel rendering, to the point that “voxel rendering” almost always refers to rendering the isosurface of a voxel grid. The isosurface of a voxel grid is the surface present between voxels that surpass some threshold and those which don’t (i.e. the boundary between voxels that are “present” and those which aren’t). To render such an isosurface, the approach varies on the underlying render technique used. Such render techniques generally fall into two categories: rasterization and ray-tracing.

Ray-tracing is the earliest method of rendering, which works by simulating rays of light in a scene, making it capable of rendering realistic images. However, it is computationally expensive, leading to the popularity of another method for real time applications: rasterization. Rasterization works by projecting triangles of a polygonal mesh onto the camera’s 2D view and filling in their bounds using small programs called “shaders”. Because of the strengths and weaknesses of both methods, voxel rendering techniques fall into two categories as well: isosurface mesh extraction for rasterization and voxel traversal for ray-tracing (there are also hybrid techniques, but they exist as an optimization of voxel traversal).

Isosurface mesh extraction, popularly known as “meshing”, is the process of converting voxels to a mesh of the isosurface, and there are many meshing algorithms, such as “greedy meshing”, “marching cubes”, “naive surface nets”, etc. All have their own characteristics making them suitable for particular applications. However, when rendering large scenes, meshing and rasterization typically struggle to scale in performance. Meanwhile, ray tracing is capable of scaling in performance, due to optimizations that are relatively easy to implement with ray tracing, such as acceleration structures and LOD (level-of-detail), and the lack of a mesh generation step when updating voxels, as well as better memory scaling (storing a mesh takes up memory on top of storing the voxel volume). Because of this difference in scaling potential, when rendering large scenes on mid to high end consumer computers, ray tracing becomes more performant than rasterization.

A. Problem Statement

The goal of this project is to implement a CPU voxel ray-tracer in the Rust language with two storage backends: a dense array storage and a sparse voxel octree storage. The dense array represents a naive voxel storage implementation, while the sparse octree represents an acceleration structure implementation: an optimization for ray traversal via a space partitioning structure. Using both storage solutions, we can

assess the benefits of octrees for optimizing ray traversal in voxel scenes.

II. METHODOLOGY

By making the ray tracer generic over storage backends, we can swap between ray tracing implementations. To assess the performance of each implementation, the benchmarks will consist of measuring the time it takes for each implementation to render a scene for a given scene size and frame resolution.

III. IMPLEMENTATION

The program is divided into the following parts:

- Voxel Generator
- Scene Object with Swappable Storage Backends
 - Dense Array-Backed Storage
 - Sparse Octree-Backed Storage
- Framebuffer
- Camera Controller
- Ray Tracer
- Image Exporter
- Command Line Interface

[Elaborate on structure]

A. Voxel Generation

[Elaborate on voxel generation]

B. Scene Object

[Discuss the scene object]

1) *Scene trait*: [Elaborate on scene trait]

2) *Ray struct*: [Elaborate on ray struct]

3) *IAabb struct*: [Elaborate on aabb struct] An AABB describes the position and extents of a volume. By iterating over every coordinate in the AABB, the storage backend can query the voxel generator for a voxel and store it if present.

[Discuss fast intersection tests] The AABB can also be used during a ray trace, as a quick intersection test for the entire volume.

C. Dense Storage

[Dense: Fast Traversal Algorithm]

For the dense storage backend, there exist many algorithms for ray tracing through a multidimensional array, but most are based on the “Fast Voxel Traversal” algorithm by Amanatides and Woo, which will be what we use for ray tracing in the dense storage backend [1]. This algorithm works by tracking the index and position for each axis and moving the ray along the axis with the shortest distance to the next index. By doing so, it ensures that no voxels are missed during iteration. When a voxel is found or the indices go out of bounds, it returns.

D. Sparse Storage

[Sparse: Octree space partitioning]

In the world of 2D graphics there is a data structure known as a quadtree which is used to divide a 2D plane into 4 quadrants one to represent top left, top right, bottom left, and bottom right. These quadrants are stored as children of a parent node and so on and so forth. An Octree is the 3D equivalent of this. Within each Octree is a root that represents an area in 3D space, within this root there are 8 children which are 8 subdivisions of the parent root, each subsequent child breaks down the space until the children being stored are individual voxels.

An octree is a helpful tool in rendering 3D spaces as they are a very compact way of storing voxels without wasting memory. They are also very quick to traverse as one can navigate to a point in a 3D space by either descending or ascending the structure to pinpoint any given location. One of the key features is the PreOrder Traversal, this is helpful because it can help to determine line of sight for an object, or for our purposes, a ray. Since the ray can check what objects it will or will not collide with by simply traversing the octree it greatly reduces a lot of redundant calculations that would have been performed were the voxels to be stored in a different data structure. The key difference between Dense and Sparse storage however is that

E. Framebuffer and Image Exporter

Once we have processed all the scene information, it is time to save it to an image so we can view it. We do this by adding information to a framebuffer, then copying the framebuffer into a PNG file. The framebuffer is a data structure that holds all the image information of the rendered scene. It contains a memory reference to a heap-allocated array of 4-byte integers for each pixel. Each integer represents the RGBA values of a pixel, with the first byte holding the R value, second byte holding the G value, etc. The array uses an atomic data type, which allows for multiple threads to access and modify the values of different pixels safely and concurrently to allow faster image generation. We use Rust’s image crate to then write to a PNG file.

F. Camera Controller

[Elaborate on camera and ray instantiation]

G. Ray Tracer

[Elaborate on how ray tracer brings parts together]

H. Command Line Interface

[Elaborate on the CLI]

IV. EVALUATION

[Discuss evaluation via benchmarks]

[Discuss analysis via tracing]

REFERENCES

- [1] Amanatides, John and Woo, Andrew. (1987). A Fast Voxel Traversal Algorithm for Ray Tracing. Proceedings of EuroGraphics. 87.