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 "**pic**ture **el**ement" that represents part of a picture, a "voxel" is a "**vo**lume **el**ement" 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

We have achieved the goal for this project: implement a CPU voxel ray-tracer in the Rust language with two storage backends: a dense array storage and a sparse voxel octree storage, and benchmark them against each other. 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

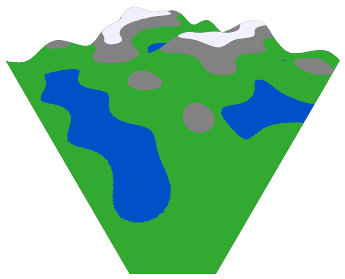

Fig. 1. A render produced by our voxel ray-tracer.

can assess the benefits of octrees for optimizing ray traversal in voxel scenes via benchmarks simulating rendering scenes at different sizes and resolutions. See Fig. 1 for an example render produced by our voxel ray-tracer.

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

The clear boundaries between components was beneficial to dividing work among colleagues.

A. Voxel Generation

Voxels are generated by the VoxelGenerator module. This module procedurally generates voxel data using Perlin noise. Perlin noise is a gradient-based noise function developed by Ken Perlin used in procedural generation due to its natural, smooth variation [2] as seen in Figure 2. The algorithm outputs pseudo-random values that transition smoothly across space, avoiding the harsh discontinuities of pure random noise.

The module defines key structures and functions to determine voxel color and existence based on height-derived terrain. The module consists of the following structures and functions:

1) `Voxel` *struct:* The `Voxel` struct encapsulates data for a single voxel. It contains one field:`color` of type `U8Vec3`: A 3-component vector representing RGB color values.

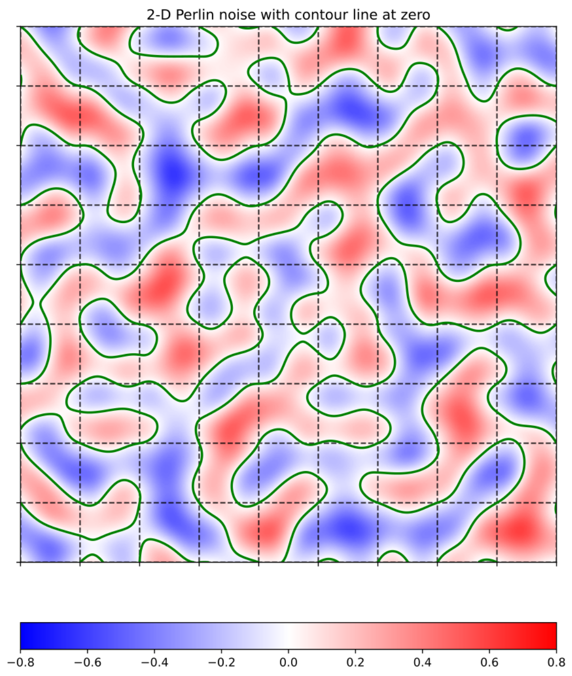

Fig. 2. Example output of 2-D Perlin noise

2) `VoxelGenerator` *struct:* The `VoxelGenerator` struct is responsible for generating voxel data using Perlin noise. It contains `perlin`: An instance of the `Perlin` struct, which contains the perlin noise function seeded either randomly or explicitly.

The `VoxelGenerator` can be initialized with either a random seed or a specific seed (mainly used for testing purposes)

Several constants are defined to control terrain parameters:

- `HEIGHT`: Maximum height for voxel terrain.
- `ROUGHNESS`: Roughness factor for noise.
- `SCALE`: A scaling factor combining roughness and height to normalize noise frequency.

3) `VoxelGenerator lookup()` *function:* The lookup function computes Perlin noise at a given `x,z` position by mapping the noise to a terrain height by scaling the values by the `SCALE` constant. Then, a noise value is retrieved from the `perlin.get()` function using the scaled coordinates. The output of the noise function is in the range $[-1.0, 1.0]$. This value is normalized to the range $[0, \texttt{HEIGHT}]$ to compute the terrain height. It then compares the `y` coordinate of the input position to the terrain height, and returns `Some<Voxel>` if the position is at or below the terrain; otherwise returns `None`.

4) `VoxelGenerator height_to_color()` *function:* The function maps a height value to one of four constant RGB colors depending on the normalized terrain height, simulating terrain types like water, grass, mountains, and snow.

B. Scene Object

The scene object stores the scene and provides an interface for ray tracing. Our project has two scene object implementations: a dense array-backed storage object and a sparse octree-backed storage object. There is some overlap in the implementations of these objects, particularly the `Scene` trait, `Ray` struct, and `IAabb` struct:

1) `Scene` trait: The `Scene` trait has two items: a function for creating the scene object given the voxel generator and a method for tracing a ray through the scene. This is the interface from which the ray tracer interacts with the scene.

2) `Ray` struct: The `Ray` struct is a vector pair of the ray origin and the ray direction. It is used to simulate light rays that traverse the scene from the camera.

3) `IAabb` struct: The `IAabb` struct represented an axis-aligned bounding box (AABB) represented by integer vectors. By iterating over every coordinate in an AABB, the storage backend can query the voxel generator for a voxel and store it if present.

An AABB can also be used during a ray trace, as a quick intersection test for the entire volume. The algorithm used for fast intersection tests is from ”An Efficient and Robust Ray-Box Intersection Algorithm” by Williams et al. [1] To summarize how the algorithm works: it takes advantage of properties observed in intersection between rays and AABBs to quickly check if the ray crosses the box boundaries.

A more detailed but brief explanation is that the algorithm projects boundary line intersections to a 1D line representing distance traveled across the ray and checks the order of these intersections to verify that the ray enters all bounds before exiting any other bounds (See Fig. 3 for an example of the calculations).

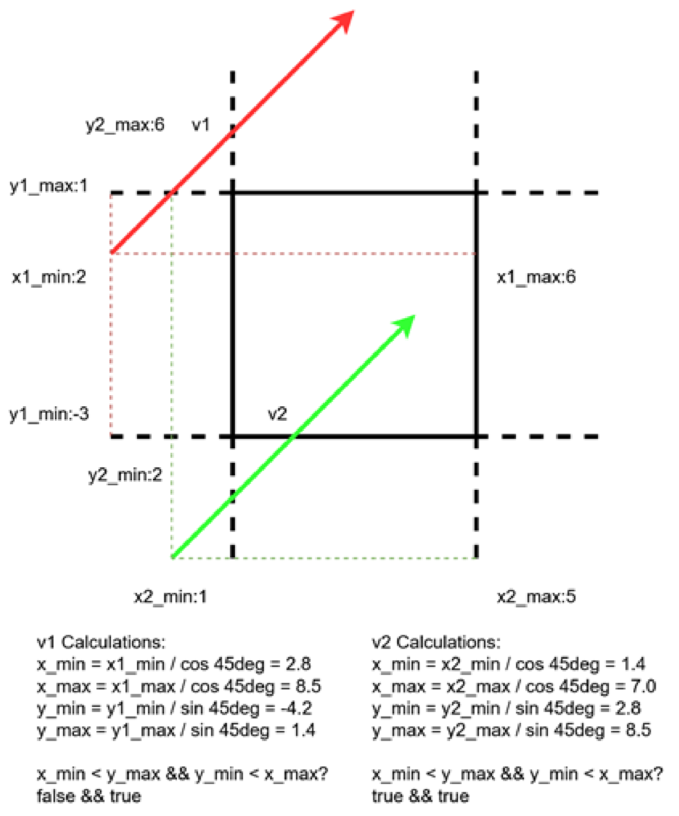
y2_max:6
v1
y1_max:1
x1_min:2
x1_max:6
y1_min:-3
v2
y2_min:2
x2_min:1
x2_max:5
v1 Calculations:
x_min = x1_min / cos 45deg = 2.8
x_max = x1_max / cos 45deg = 8.5
y_min = y1_min / sin 45deg = -4.2
y_max = y1_max / sin 45deg = 1.4
x_min < y_max && y_min < x_max?
false && true
v2 Calculations:
x_min = x2_min / cos 45deg = 1.4
x_max = x2_max / cos 45deg = 7.0
y_min = y2_min / sin 45deg = 2.8
y_max = y2_max / sin 45deg = 8.5
x_min < y_max && y_min < x_max?
true && true

Fig. 3. Two example calculations for fast ray-box intersections using 2D coordinates.

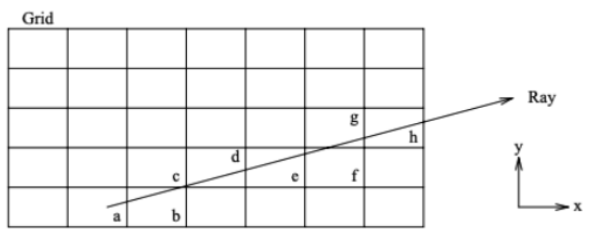

Fig. 4. The fast voxel traversal algorithm in action.

C. Dense Storage

The dense storage backend stores every possible position in the voxel volume in a single array. It is most beneficial to use such a storage solution when most positions hold present voxels, or in other words, is densely populated, hence the name.

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 [3]. 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 (See Fig. 4). By doing so, it ensures that no voxels are missed during iteration while not testing positions that don’t intersect the ray. When a voxel is found or the indices go out of bounds, it returns.

D. Sparse Storage

The sparse storage backend stores present voxels in a space partitioning structure called an ”octree” (See Fig. 5 for examples). It is most beneficial to use such a storage solution when there are large gaps in the volume, or in other words, when the volume is sparsely populated, hence the name.

Octrees are tree structures that have branches that subdivide space along each axis, allowing for traversing over ”octants”, the 8 subdivisions of each branch. The benefit is that octants that don’t contain voxels are culled, meaning that entire areas of empty space can be skipped when a ray traverses the octree.

An easy way to understand ray traversal through octrees is to examine ray traversal for their 2D counterpart: quadtrees, which subdivides 2D space into quadrants (See Fig. 6).

When beginning ray traversal, the first quadrant to check is the one where the ray originates in the direction of (so compare the volume origin to the ray origin). If the quadrant has children, the algorithm is recursively called for that quadrant. Otherwise, other quadrants are checked if the ray crosses the

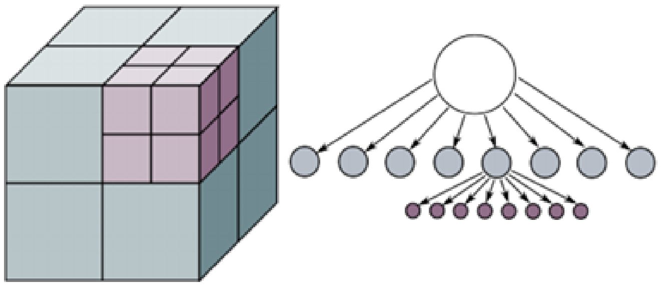
Fig. 5. Two example visualizations of an octree.

respective local origin's axes (the local origin is the origin of the volume for the top level and origin of quadrants for recursive calls) and in order of the distance required to cross the local origin.

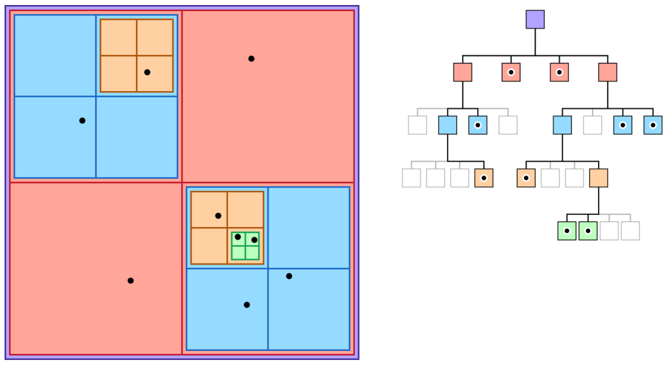
Fig. 6. Two example visualizations of a quadtree (note: this example shows variable depth for storing items, while our octree implementation has a constant depth).

In order to visualize octree traversal, a debug mode was added to the ray tracer, which renders the edges of bounding boxes and colors voxels black (See Fig. 7).

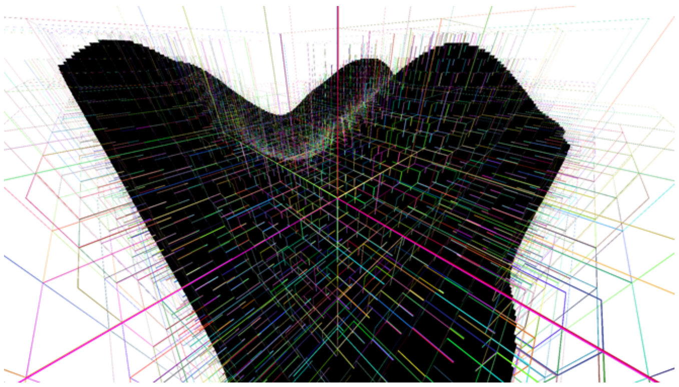
Fig. 7. An octree structure visualization for debugging.

It renders the edges by running face intersection tests for the ray and checking if any of the intersection points are within some threshold (that is, if two points on separate faces are close to each other, then they must lie near the edge that joins both faces). The color for the edge is picked via Pearson hashing, a simple hashing algorithm for small inputs that also produces byte-sized hashs via a lookup table (See Alg. 1 for the generic algorithm). The algorithm was modified to consume bytes from a position vector and produce three bytes corresponding to color channels.

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 an image file specified by a given output path.

F. Camera Controller

Any image of the scene must be viewed from somewhere and at some angle; this is handled by the `Camera` controller. The `Camera` holds several attributes corresponding to the camera's position, orientation, field of view, and related values. From these, other values such as the distance in the scene between each pixel, the location of the top-left pixel in the scene, and the boundaries of the visible area are also calculated. The `Camera` class contains a `get_ray` function that receives pixel coordinates and creates a `Ray` object from the camera's position into a direction corresponding to these coordinates.

In further detail, when creating a Camera object, there are seven parameters the user may specify; otherwise, default values are used. They are as follows:

- `img_width`: The width of the image in pixels.
- `img_height`: The height of the image in pixels.
- `vertical_fov`: The vertical field of view in degrees.
- `lookfrom`: The 3D coordinates of the camera.
- `lookat`: The 3D coordinates of the point the camera is pointed towards.
- `cam_up`: A vector representing the ”up” direction relative to the scene.
- `focus_dist`: Distance from the camera to the focus point.

From these, the program then converts the `vertical_fov` from degrees to radians, then calculates the height angle `h`, multiplying it by `2 * focus_dist` to get the `viewport_height`. Then, it uses the ratio of the image dimensions, `img_width` and `img_height`, to calculate the `viewport_width`.

It also creates an orthonormal basis for the camera through a series of cross products:

- First, the forwards-vector (z-axis), `w` is calculated by normalizing a vector from `lookfrom` to `lookto`.

- Then, the right-vector (x-axis), `u` is calculated by taking the cross-product between `w` and `cam_up` and normalizing.
- Finally, the up-vector (y-axis), `v` is calculated by taking the cross-product between `u` and `w`.

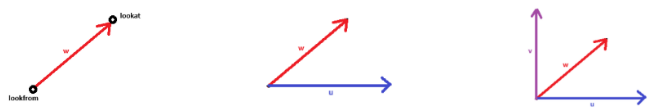

Fig. 8. The orthonormal basis after each step.

The vectors `viewport_u` and `viewport_v` are then calculated by multiplying `u` by `viewport_width` and `v` by `viewport_height`, respectively. Dividing them by the image dimensions, we get `pixel_delta_u` and `pixel_delta_v`, representing the horizontal and vertical distances between each pixel. Then, we subtract half of the `viewport_width` and `viewport_height` from the center and add half of `pixel_delta_u` and `pixel_delta_v` to get the scene location of the top-left pixel, `pixel00_loc`.

These calculated values are then used in the `get_ray` function as follows: The function takes in the pixel coordinates, `i` and `j`; it calculates `pixel_sample` by starting from the top-left pixel `pixel00_loc` and adding `i * pixel_delta_u` and `j * pixel_delta_v` to it.

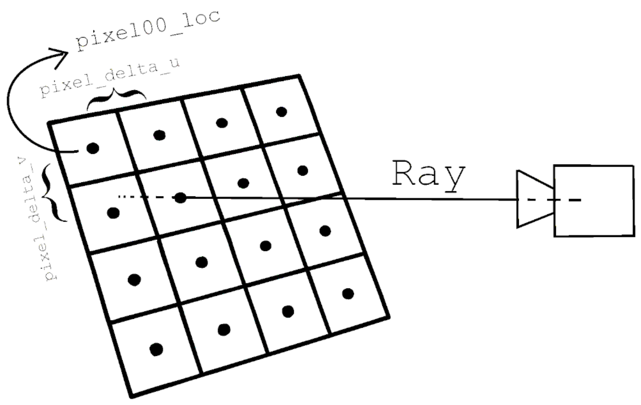

Fig. 9. The `get_ray` function, visualized. In a 4x4 pixel image, this ray goes through the pixel in the second row, second column.

Then, it returns a `Ray` object with origin in `center` and direction going from there into `pixel_sample`.

G. Ray Tracer

The ray tracer brings together the scene object, camera, and framebuffer components to provide an render method that generates a new framebuffer with contents rendered from the scene object with rays instantiated by the camera. The ray tracer runs in parallel with help from the `rayon` crate, which provides traits for parallel iterators. `rayon`'s parallel iterators work by splitting an iterator across threads via work-stealing, so if a thread finishes it's work before another, it can split the iterator on another thread to keep itself busy.

What the ray tracer needs to iterate over is the pixel coordinates and values, in order to instantiate rays using the coordinates and save the result to the value. To do this, we implemented a parallel iterator for the framebuffer that produces `PixelRef` objects, that store the coordinates and references to the pixel value.

This requires implementing three components: `ParIter` (our parallel iterator), `ParIterProducer` (our iterator producer, which is able to split itself and convert into an iterator), and `Iter` (our normal iterator which individual threads run). The flow of information can be roughly modeled as `Framebuffer` → `ParIter` → `ParIterProducer` → `Iter` → `PixelRef`.

For each `PixelRef` we run a task which instantiates a ray from the camera using the pixel coordinates, calls the `trace` method on the scene object with that ray, and saves the result to the value reference, after first converting the color vector to an unsigned integer.

H. Command Line Interface

To properly interact with our program a CLI(Command Line Interface) was developed using the CLAP(Command Line Argument Parser) library. CLAP accepts flags and corresponding arguments to augment or modify how the scenes are generated. We implemented 7 key flags:

- Backend (dense or sparse)(string)
- Size (int)
- Position (x,y,z)(int)
- Seed (int)
- Out (string)
- Width and Height (int)
- Debug (boolean)

The *Backend* flag is used to swap between dense and sparse storage, this helped us to quickly compare the two modes of rendering and collect data about rendering speeds, performance, and file size.

The *Size* flag is used to determine the size of the scene being generated, this is also useful in benchmarking the two different backends as the increase in speed between the two is dependent on size.

The *Position* flag is used to move the camera around in a 3D space, hence why it takes in x,y, and, z coordinates, this allows the user to get a better view of the scene and examine the behaviors of different seeds.

The *Seed* flag is used to input or, if left blank, randomize, the seed for what is being rendered, it determines where voxels are positioned which in turn determines how they are colored according to their height.

The *Out* flag is used to designate where the image of the render is stored by inputting a file path, if not, the default location is 'render.png'.

The *Width and Height* flags determine the resolution of the image, this is also useful for benchmarking the two different methods of generation as resolution can have an effect on render time.

The *Debug* flag is used to enable or disable the debug mode which is crucial for building on top of what we have already developed and for troubleshooting any potential issues that may arise.

Overall the Command Line was not the most complex component of the *Voxel Ray Tracer* but it was a crucial tool for us to be able to interact with the renderer and take benchmarks for how different settings performed.

IV. Evaluation

To evaluate the performance benefits of sparse voxel octrees over array-backed storage, we constructed benchmarks via `criterion.rs` that measured the performance of each storage backend over scene size and frame resolution. The benchmarks ran across 12 threads on a mid-range consumer PC with an AMD Ryzen 5 2600 6-core CPU and 32GB of RAM.

TABLE I
1080p Resolution Benchmarks

Scene Size	Dense	Sparse	Ratio
50	37.557 ms	211.86 ms	0.1773
100	192.48 ms	247.86 ms	0.7766
250	811.93 ms	122.55 ms	6.6253

TABLE II
4k Resolution Benchmarks

Scene Size	Dense	Sparse	Ratio
50	261.11 ms	2.5230 s	0.1035
100	3.0203 s	3.5526 s	0.8502
250	10.016 s	2.0995 s	4.7706

The benchmarks assessed the storage backends for small and medium scenes, with sizes of 50, 100, and 250 voxel extents (the total volume is $(2x)^3$ voxels for each x size), and for resolution of 1080p (1920×1080 pixels) and 4k (7680×4320 pixels). We only tested small and medium sized scenes since the time to render larger scenes took too long to collect samples for in a practical time for iterating modifications. The camera for each scene was located in the top corner of the scene, facing inwards toward the origin (from a position of $(x - 10)\hat{v}$). The results are the average time to render a scene, for 100 samples.

For small scenes, the dense storage solution was faster, but that quickly changed as scenes grew in size. For scenes with size 250, the sparse storage solution was incredibly fast, even faster than it was for smaller scenes. This was due to the voxel generation having a height limit of 100, which meant for larger scenes that there was more empty space. Since the sparse backend benefits from empty space, it was able to skip more empty space, leading to faster render times.

See Fig. 10 for violin plots.

A. Tracing

To allow for further optimization, tracing instrumentation was added to the program with the `tracing`, `tracing-subscriber`, and `tracing-tracy` crates, which allow for timing functions using the Tracy profiler. From that, we can see where the program spends the most time in and look for ways of shaving time off in those areas (See Fig. 11 for an example).

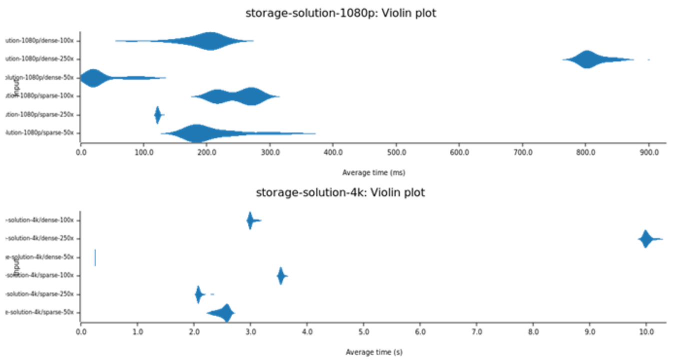

Fig. 10. Violin plots of the benchmarks.

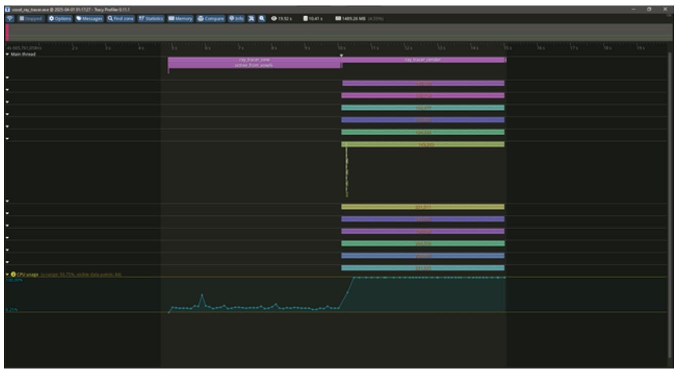

Fig. 11. An example of collected traces (notice that half the time of the program is spent constructing the scene on a single thread).

Generating traces is costly, so this feature was put behind a feature flag.

V. Conclusion

In this paper, we aimed to compare two different storage solutions for a CPU voxel ray-tracer implemented in Rust. By comparing the average time it took to render randomly-generated scenes of varying sizes and at different resolutions, we found that a naive implementation (storing every voxel in an array, traversed with Amanatides and Woo's "Fast Voxel Traversal" algorithm) was faster than a sparse storage solution (storing every voxel in an octree structure, traversed with a recursive algorithm) at smaller scene sizes, but scaled poorly and was significantly slower for the largest scenes that we tested. The sparse storage solution was faster for large scenes because its recursive structure eliminates some unnecessary traversal over empty space.

Some limitations and future areas of research for our study include but are not limited to the following: we only tested the ray tracer with relatively small scenes (¡250 voxel extents), as it would have taken an impractical amount of time to do so with our computing resources. We also did not study the effects of using different visual textures.

References

[1] Williams, Amy & Barrus, Steve & Morley, R. & Shirley, Peter. (2005). An Efficient and Robust Ray-Box Intersection Algorithm. J. Graphics Tools. 10. 49-54. 10.1145/1198555.1198748.

[2] Perlin, Ken (1985). An image synthesizer. ACM SIGGRAPH Computer Graphics. 19 97–8930: 287–296. 10.1145/325165.325247

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

Algorithm 1 Generic Pearson Hashing Algorithm

```
Input: Byte array $C$
Output: Byte-sized hash $h$
    Initialisation : $T$ is a permutation of an array containing
    bytes 0 to 255
1: $h \leftarrow 0$
2: for each $c \in C$ do
3:     $h \leftarrow T[h \oplus c]$
4: end for
5: return $h$
```