The University of Manchester

Building a Ray-Traced Rendering Engine on Sparse Voxel Grids

by
Aaron-Teodor Panaitescu
supervised by
Prof. Steve Pettifer

BSc. (Hons) in Computer Science
The University of Manchester
Department of Computer Science

Year of submission
2024

Student ID
10834225

Contents

Word count: 12,524

List of figures

List of tables

Abstract

Rendering engines are critical in computer graphics, serving as the backbone of visualisation for digital media, including video games, simulations, virtual reality, and animated films. These engines are responsible for converting 3D models, textures, and lighting information into the compelling images users see on their screens. A rendering engine's efficiency and capabilities directly influence the digital experience's realism and interactivity.

This thesis explores the development and implementation of a ray-traced voxel rendering engine utilising sparse grids to enhance real-time rendering capabilities. The focus is on leveraging sparse data structures to manage volumetric data efficiently, thus allowing for intricate rendering details and high performance. The research investigates various techniques in ray tracing to optimise the rendering process. The outcomes demonstrate that sparse voxel grids combined with discrete signed distance fields can significantly reduce memory usage while maintaining rendering speed. This work contributes to computer graphics and game development by providing insights into the application of sparse data structures in real-time rendering environments.

Declaration of originality

I hereby confirm that this dissertation is my own original work unless referenced clearly to the contrary, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual property statement

i The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made *only* in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations (see http://www.library.manchester.ac.uk/about/regulations/_files/Library-regulations.pdf).

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Steve Pettifer, for his guidance, encouragement, and expertise throughout this research project. My thanks also go to my partner Diana, whose support and understanding have been my anchor in my academic pursuits. I also extend my deepest appreciation to my parents, whose love and belief in my potential have been my constant motivation.

Part I

Introduction

1 Motivation

The field of computer graphics has witnessed substantial evolution, driven by countless advancements in technology and an ever-growing demand for more realistic and interactive digital experiences. Rendering engines, fundamental to this progression, are the drivers behind compelling visual content in various applications, ranging from cinematic visual effects to complex scientific visualisations and immersive video games.

Over the last decade, advancements in hardware, particularly GPUs, have significantly expanded the capabilities of graphical applications. Modern GPUs offer tremendous computational power, enabling more complex calculations at higher speeds. This hardware evolution has made real-time rendering, particularly ray tracing, more feasible for widespread use. Thanks to these advancements, ray tracing, once limited to pre-rendered scenes due to its computational intensity, can now be performed in real time. This shift has revolutionised gaming and interactive media, allowing for cinematic-quality graphics during gameplay.

This project aims to push the boundaries of what can be achieved with voxelised real-time rendering by leveraging efficient sparse voxel data structures. The goal is to develop a tool that supports the creation of visually interesting digital environments and contributes to the ongoing research and development in rendering technologies.

This project, therefore, stands at the intersection of theoretical exploration and practical application. It aims to harness the power of modern hardware to solve complex rendering challenges and contribute valuable insights and tools to the field of computer graphics.

2 Objectives

1. Develop advanced ray tracing algorithms that fully utilise modern hardware.
2. Explore acceleration structures that optimise ray casting performance.
3. Develop a voxel rendering engine that integrates these advanced algorithms and structures.
4. Design the engine architecture to take full advantage of the algorithms and technology it is built on, ensuring robustness, efficiency, and rendering performance.
5. Compare and test these algorithms against each other to validate improvements in speed and quality.

3 Aims

1. *Performance:* Optimize the rendering engine to handle complex scenes with high levels of detail and dynamic changes efficiently, striving for speed and graphical output enhancements.
2. *Safety:* Make sure the system is reliable, minimize memory leaks and undefined behaviours.
3. *Cross-Platform:* Ensure the engine is not tied to a specific platform, operating system or graphics backend.
4. *Futre-Proofing:* Build the engine on a forward-looking graphics API designed to be efficient, powerful, and broadly supported.

4 Report structure

The report is comprised of 5 parts:

- Part I gives an introduction to the project and the report,
- Part II details the project background, literature review and related work,
- Part III presents a detailed walkthrough of the design and development of the project,
- Part IV shows the results and experiments of the project,
- Part V presents the project conclusions.

Part II

Background and Literature Review

5 Rendering engines

Graphics engines are the core software components responsible for rendering visual content in applications ranging from video games to scientific simulations and movie visual effects. Engines abstract the complexities of rendering by providing developers with high-level tools and interfaces to represent digital environments.

Rendering engines have evolved from the simple wire-frame models of the 1960s to today's complex 3D systems, driven by advancements in computational power and graphical standards[1] like OpenGL introduced in the early 1990s.

5.1 Primitves

At the heart of any graphical engine is the concept of primitives, the simplest forms of graphical objects that the engine can process and render. Primitives are building blocks from which more complex shapes and scenes can be constructed.

Polygons, particularly triangles, are the most commonly used primitives in 3D graphics. This is owed to their simplicity and flexibility, allowing the construction of virtually any 3D shape through *tesselation*. Polygonal meshes define the surfaces of objects in a scene, with each polygon vertex typically associated with additional information such as colour, texture coordinates, and normal vectors for lighting calculations.

Voxels represent a different approach to defining 3D shapes; they are essentially three-dimensional pixels. Where polygons define surfaces, voxels establish volume, with each voxel potentially containing colour and density information. This characteristic makes voxels particularly well-suited for rendering scenes with materials that have intricate internal structures, such as fog, smoke, fire, and fluids.

5.2 Ray-tracing vs. Rasterization

Rendering engines can utilise two main rendering techniques for rendering scenes: ray tracing and rasterisation. Both have advantages and trade-offs.

Rasterisation is the most widespread technique used in real-time applications. It converts the 3D scene into a 2D image by projecting vertices onto the screen, filling in pixels that makeup polygons, and applying textures and lighting. Over the development of the graphics programming industry, graphics hardware has become extremely efficient at performing rasterisation, making it the standard for video games and interactive applications.

Ray-Tracing, in contrast, simulates the path of light as rays travelling through a scene to produce images with realistic lighting, shadows, reflections, and refractions. Ray tracing is computationally intensive but yields higher-quality images, making it favoured for applications where visual fidelity is critical. However, recent advancements in hardware have begun to bring real-time ray tracing to interactive applications.

Ray tracing, conceptualised by Arthur Appel in 1968[2], offers photorealistic images by simulating light paths, but its computational intensity limited early use to non-real-time applications. Rasterisation, popularised in the 1970s and optimised by GPU advancements, became the standard for real-time graphics, though *recent* hardware innovations are now enabling real-time ray tracing.

6 Representing voxels

Various data structures can be employed to represent and manipulate voxels in program memory efficiently. Each method entails trade-offs between memory usage, access speed, and implementation complexity. Access speed refers to the time complexity of querying the data structure at an arbitrary point in space to retrieve a potential voxel.

6.1 Voxel grids

A voxel grid is the most straightforward and intuitive approach to representing volumetric data. The 3D space is divided into a regular grid of voxels, each holding information such as colour, material properties, or density. This method provides direct $O(1)$ access to voxel data.

However, this simplicity comes at a significant disadvantage: memory consumption. As the bounding volume or the level of detail of the scene increases, the memory required to store the voxel grows by $O(N^3)$. Additionally, empty space can occupy a majority of the memory space. For example, consider a scene with two voxels a million units apart in all axes. A voxel grid would have to store all the empty voxels in-between; 10^{18} memory units reserved, 2 of which carry useful data. This limitation makes the naive voxel grids impractical for large or highly detailed scenes.

6.2 Hierarchical voxel grids (N-trees)

Hierarchical grids, such as octrees, are employed to mitigate these issues. An octree is a tree data structure where each node represents a cubic portion of 3D space and has up to eight children. This division continues recursively, allowing for varying levels of detail within the scene: larger volumes are represented by higher-level nodes, while finer details are captured in lower levels.

The primary advantage of using an octree is spatial efficiency. Regions of the space that are empty or contain uniform data can be represented by a single node, significantly reducing the memory footprint. Furthermore, octrees facilitate efficient querying operations, such as collision detection and ray tracing, by allowing the algorithm to discard large empty or irrelevant regions of space quickly.

Hierarchical grids introduce complexity in terms of implementation and management. Operations such as updating the structure or balancing the tree to ensure efficient access can be more challenging than those of uniform grids. Another sacrifice is access time, as querying an arbitrary region of space can entail walking down the tree for several levels. Nonetheless, the benefits of hierarchical representations often outweigh these drawbacks for applications requiring large, detailed scenes with a mix of dense and sparse regions. Therefore, N-trees are frequently used in voxel engines.

Donald Meagher introduced the concept of octrees in 1980[3] as a means to manage spatial data in 3D computer graphics efficiently. This technique quickly became integral in applications like 3D rendering and geometric modelling, where it revolutionized spatial data optimization by balancing detailed representation with computational efficiency.

6.3 Volumetric Dynamic B+tree

Volumetric Dynamic B+tree or **VDB** was introduced in 2013 by Ken Museth[4] from the DreamWorks Animation team.

> It is a Volumetric, Dynamic grid that shares several characteristics with B+trees. It exploits spatial coherency of time-varying data to separately and compactly encode data values and grid topology. VDB models a virtually infinite 3D index space that allows for cache-coherent and fast data access into sparse volumes of high resolution.

At its core, VDB functions as a shallow N-tree with a fixed depth, where nodes at different levels vary in size. The top level of this tree structure is managed through a hash map, enabling VDB models to cover extensive index spaces with minimal memory overhead. This design achieves $O(1)$ access performance and effectively stores tiled data across vast spatial regions.

The VDB data structure was introduced along with several algorithms that fully use the data structure's features, offering significant improvements in techniques for efficiently rendering volumetric data. These are some of VDB's benefits, as detailed in the original paper.

1. *Dynamic.* Unlike most sparse volumetric data structures, VDB is developed for both dynamic topology and dynamic values typical of time-dependent numerical simulations and animated volumes.
2. *Memory effcient.* The dynamic and hierarchical allocation of compact nodes leads to a memory-efficient sparse data structure that allows for extreme grid resolution.
3. *Fast random and sequential data access.* VDB supports fast, constant-time random data lookup, insertion, and deletion.
4. *Virtually infinite.* VDB, in concept, models an unbounded grid in the sense that the accessible coordinate space is only limited by the bit-precision of the signed coordinates.
5. *Efficient hierarchical algorithms.* The B+tree structure offers the benefits of cache coherency, inherent bounding-volume acceleration, and fast per-branch (versus per-voxel) operations.

These benefits make VDB a very compelling data structure that serves as the building block of a voxel-based rendering engine.

7 Ray tracing

To render a scene using ray tracing, camera rays are shot through the view frustum and into the scene. At each object intersection, part of a ray is absorbed, reflected, and refracted. To achieve realistic results, a rendering engine needs to model as many of these light interactions as possible in each frame's time budget.

This section delves into integrating ray tracing within the graphics pipeline and the methods used to implement it, focusing on casting a ray through a scene.

7.1 Graphics pipeline

The graphics pipeline of a rendering engine is the underlying system of a rendering engine that transforms a 3D scene into a 2D representation that is then presented on a screen. While rasterization transforms 3D objects into 2D images through a series of stages (vertex processing, shape assembly, geometry shading, rasterization, and fragment processing), the ray tracing pipelines introduce a paradigm shift. It primarily involves calculating the path of rays from the eye (camera) through pixels in an image plane and into the scene, potentially bouncing off surfaces or passing through transparent materials before contributing to the colour of a pixel.

Calculating a ray's path is central to ray tracing, so the performance of the algorithm that does this calculation is critical.

7.2 Casting a ray

Ray casting techniques vary depending on the representation of the 3D world within the rendering engine. This section introduces basic ray casting techniques, while subsequent discussions cover methods specific to voxel-based environments.

Ray marching

A straightforward way to represent a 3D environment would be a mathematical function of sorts. It would take the coordinates of a point as input and return the material's properties at that point (provided an object is present).

The first algorithm one might develop when trying to cast a ray through an unknown scene is ray marching. It involves incrementally stepping along a ray, sampling the scene for collisions at each step. The chosen step size must be sufficiently small to ensure no detail is missed.

While simple, ray marching has drawbacks, especially in terms of performance. Considering the need to process millions of pixels per frame within the time constraints of high frame rates, it becomes apparent that iterating a ray tens of thousands of times for every pixel is impractical for modern engines.

These constraints require exploring more advanced techniques to meet the goal of visual realism and performance.

Ray casting

A 3D environment could also be represented as a collection of polygons that form meshes.

Ray casting finds the intersection of rays with geometric primitives (e.g. triangles and circles). This method skips stepping along the ray entirely by using the underlying mathematics of intersecting lines with polygons.

The fundamental issue with this approach is that rays must be checked for an intersection with all the primitives in the scene. Thus, computing a single ray's intersection has linear complexity in terms of the number of polygons in the scene.

Signed Distance Fields

Signed distance fields (SDF) are a different way of representing the environment. A SDF provides the minimum distance from a point in space to the closest surface, allowing the ray marching algorithm to skip empty space and efficiently determine surface intersections. With the distance to the nearest surface known, ray marching can be performed by stepping along the ray with that distance, drastically reducing the number of steps needed to cast a ray.

Combining SDF with ray marching offers a powerful method for rendering complex scenes, including soft shadows, ambient occlusion, and volumetric effects. This combination is highly flexible and can create highly detailed and intricate visual effects, particularly in procedural rendering and visual effects.

SDFs are not without drawbacks. They can be difficult to maintain and computationally expensive to generate or update. In practice, distance data cannot be of arbitrary size, as that distance information comes at the cost of program memory.

SDFs have been used in real-time rendering, usually in a raymarching context, starting in the mid-2000s. In 2007, Valve used SDFs to render large pixel-size smooth fonts on the GPU in its games[5].

7.3 Casting a ray on a voxel grid

The ray casting methods presented so far do not take advantage of the discrete voxel grid on which this rendering engine is based. This section presents efficient algorithms that can use the underlying representation of a hierarchical voxel grid.

Digital Differential Analyzer

Basic ray marching can be improved on a discrete voxel grid by stepping from voxel to voxel. Because voxels are the smallest unit of space, a ray can safely step from one to the next, ensuring there is nothing else in between.

The Digital Differential Analyzer (DDA) line drawing algorithm does precisely that; it marches along a ray from voxel to voxel, skipping all space in between.

DDA works by breaking down the minimum distance a ray travels to intersect a grid line on each axis. At each iteration, it steps to the closest grid intersection along the ray.

Hierarchical Digital Differential Analyzer

On a hierarchical grid, the DDA algorithm can take advantage of the data structure's topology by stepping through empty, larger chunks. A ray cast using HDDA essentially performs DDA at the level in the tree it is currently at.

Ken Museth introduced a version of the HDDA algorithm for the VDB data structure in 2014[6]. This algorithm can be highly efficient; large empty areas can be skipped in a single step, drastically reducing the required steps to march a ray.

8 Summary of similar systems

8.1 OpenVDB[7]

> "OpenVDB is an Academy Award-winning open-source C++ library comprising a novel hierarchical data structure and a suite of tools for the efficient storage and manipulation of sparse volumetric data discretized on three-dimensional grids. It was developed by DreamWorks Animation for use in volumetric applications typically encountered in feature film production and is now maintained by the Academy Software Foundation (ASWF). "

This is the original library developed by the authors of the VDB paper. It is very well integrated with a range of rendering engines, such as Blender, Maya, and Houdini. OpenVDB is more of a library to interact with the VDB data structure than a rendering engine. It implements highly optimized algorithms for VDB data (e.g., CUDA HDDA).

8.2 All is Cubes[8]

> "This project is a game engine for worlds made of cubical blocks ("blocky voxels"). The particular features of this engine are that each ordinary block is itself made out of blocks, and all game mechanics are defined by data within the world that can be interactively edited."

This is a voxel rendering engine made on the same backend as this project, Rust and wgpu, however it employs a more standard apprach of rendering, generating triangle meshes from voxel data and performing rasterization.

8.3 Unique Contribution of this Project

To the best of my knowledge, there is currently no fully operational voxel rendering engine built solely on the VDB data structure. This project aims to fill that gap by developing a comprehensive rendering engine based entirely on VDB, leveraging its capabilities to efficiently handle complex and detailed volumetric data. Unlike other systems, which may integrate VDB as one of many components or use it for specific functions, this engine is designed to utilize VDB as the core framework for all rendering tasks. This distinction sets the project apart, offering a new perspective and opportunities to explore novel ray-tracing techniques.

Part III

Methodology

This section outlines the implementation details of the voxel rendering engine, starting with the selection of programming languages and libraries, reviewing the engine's architecture, and diving deep into the data structures and algorithms employed. It particularly focuses on VDB for voxel representation and the optimization of ray-casting algorithms. Finally, this section will discuss the extension of these algorithms to full-fledged ray tracing, allowing for dynamic lightning and glossy material support.

9 Language and framework

The voxel rendering engine is built using **Rust**, a programming language known for its focus on safety, speed, and concurrency[9]. Rust's design emphasizes memory safety without sacrificing performance, making it an excellent choice for high-performance applications like a rendering engine. The language's powerful type system and ownership model prevent a wide range of bugs, making it ideal for managing the complex data structures and concurrency challenges inherent in rendering engines. Thanks to this, no memory leak or null pointer was ever encountered throughout the development of this project.

For the graphical backend, the engine utilizes **wgpu**[10], a Rust library that serves as a safe and portable graphics API. wgpu is designed to run on various backends, including Vulkan, Metal, DirectX 12, and WebGL, ensuring cross-platform compatibility. This API provides a modern, low-level interface for GPU programming, allowing for fine-grained control over graphics and compute operations. wgpu is aligned with the WebGPU specification[11], aiming for broad support across both native and web platforms, using the WebGPU shading language (wgsl)[12]. This choice ensures that the engine can leverage the latest advancements in graphics technology while maintaining portability and performance.

The combination of Rust and wgpu offers several advantages for the development of a rendering engine:

1. *Safety and Performance:* Rust's focus on safety, coupled with wgpu's design, minimizes the risk of memory leaks and undefined behaviours, common issues in high-performance graphics programming. This added safety is thanks to Rust's idea of zero-cost abstractions.

2. *Cross-Platform Compatibility:* With wgpu, the engine is not tied to a specific platform or graphics API, enhancing its usability across different operating systems and devices.

3. *Future-Proofing:* wgpu's adherence to the WebGPU specification ensures that the engine is built on a forward-looking graphics API designed to be efficient, powerful, and broadly supported. It also allows the future option of supporting web platforms once browsers adopt WebGPU more thoroughly.

4. *Concurrency:* Rust's advanced concurrency features enable the engine to efficiently utilize multi-core processors, crucial for the heavy computational demands of rendering pipelines.

These technical choices form the foundation for building the voxel rendering engine. Following this, the engine's architecture is designed to take full advantage of Rust's performance and safety features and wgpu's flexible, low-level graphics capabilities, setting the stage for implementing advanced voxel representation techniques and optimized ray tracing algorithms.

10 Engine architecture

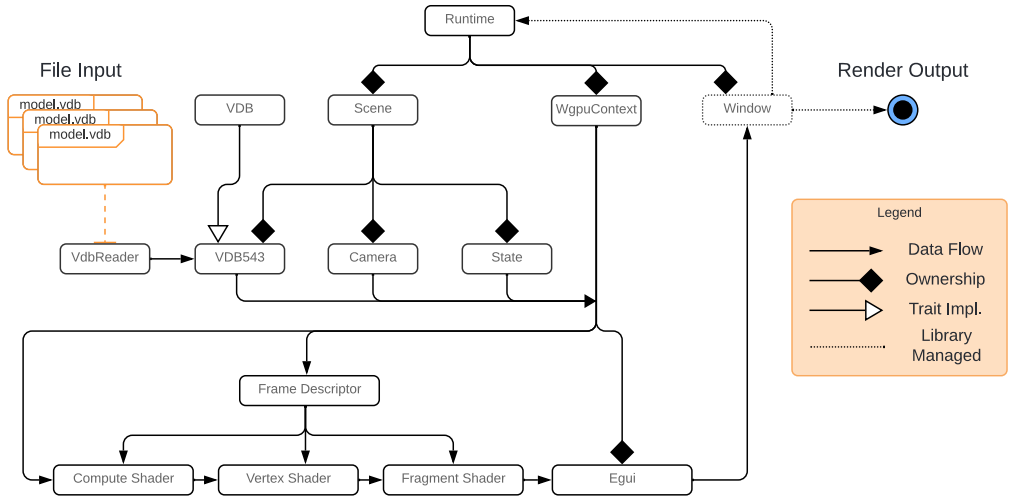


Fig. 1. Data flow from `.vdb` object file to rendering an image on the screen, dotted lines represent functionality handled by the `winit` crate.

The engine's operation centres around an event-driven main loop that blocks the main thread. This loop processes various events, ranging from keyboard inputs to redraw requests, and updates the window, context, and scene accordingly, routing each event to its corresponding handler.

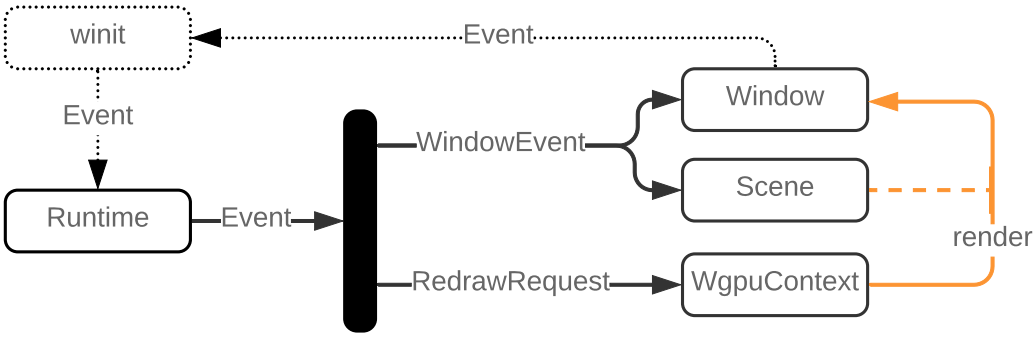


Fig. 2. Engine event-loop diagram. Dotted arrows are implemented by the `winit` crate. Black lines represent the flow of events. The orange arrow represents the the GPU context's render function called on the scene, for the window.

10.1 Runtime

At the engine's core sits `Runtime` structure, which manages the interaction between its main components:

- The `Window` is a handler to the engine's graphical window. It is used in filtering OS events that are relevant to the engine, grabbing the cursor and other boilerplate.
- The `Wgpu Context` handles the creation and application of the rendering pipeline.
- The `Scene` contains information about the camera, environment and a container voxel data structure.

```
pub struct Runtime {
  context: WgpuContext,
  window: Window,
  scene: Scene,
}
```

```
impl Runtime {
  ...
  pub fn main_loop(&mut self, event: Event, ...) {
    match event {
      ...
    }
  };
}
```

Listing 1. Runtime definition

For example, window events (e.g. keyboard & mouse input) generally modify the scene, like the camera position, and therefore are routed to the `Scene` struct.

Another key event is the `RedrawRequested` event, which signals that a new frame should be rendered. It is routed to the wgpu context to start the rendering pipeline.

The `RedrawRequested` event is emitted in `Runtime`, and when it receives the `MainEventsCleared` event, it schedules the window for a redraw.

10.2 Window

The `Window` data structure, included in the **winit**[13] crate, handles window creation and management and provides an interface to the GUI window through an event loop. This event loop is what `Runtime`'s main loop is mounted on.

The interaction between the `Window` and the `Runtime` forms an event-driven workflow. The window emits events, and the runtime manages and distributes these events accordingly, forming a feedback loop.

10.3 Scene

The `Scene` data structure holds information about the environment being rendered; this includes the model, camera, and engine state.

```
pub struct Scene {
    pub state: State,
    pub camera: Camera,
    pub model: VDB,
}
```

Listing 2. Scene definition

This section covers the camera and state implementation, and the model will be covered later section 11 when discussing the VDB implementation.

State handles information about the engine state, such as cursor state and time synchronising to decouple engine events from the FPS (e.g. camera movement should not be slower at lower FPS).

Camera describes all the elements needed to control and represent a camera:

1. *Eye:* The camera's position in the 3D space acts as the point from which the scene is observed.

2. *Target:* The point in space the camera is looking at determines the direction in which the camera is pointed.

3. *Field of View (FOV):* An angle representing the range that is in view. It refers to the implementation's FOV on the Y (vertical) axis.

4. *Aspect ratio:* The ratio between the width and height of the viewport. It ensures that the rendered scene maintains the correct proportions.

The eye and target are updated when moving the camera through a `CameraController` struct that handles keyboard and mouse input. The FOV and aspect ratio are set based on the window proportions to avoid distortion. The way in which this camera information is used will be detailed in section 10.6, which covers what information is actually sent to the GPU in compute shaders.

10.4 WgpuContext

The `WgpuContext` structure is the backbone of the rendering pipeline in the voxel rendering engine. It contains the necessary components for interfacing with the GPU using the wgpu API, managing resources such as textures, shaders, and buffers, and executing rendering commands.

Broadly, `WgpuContext` has the following responsibilities:

1. *Initialisation:* The constructor sets up the wgpu instance, device, queue, and surface. It also configures the surface with the desired format and dimensions, preparing the context for rendering.

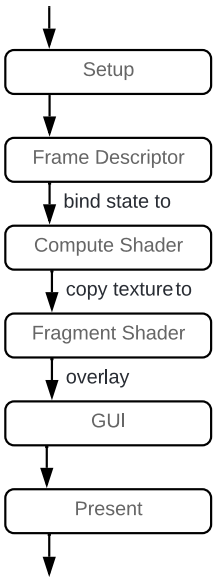
2. *Resource Setup:* The constructor prepares various resources such as textures for the atlas representation of VDB data, uniform buffers for rendering state, and bind groups for shader inputs. It also dynamically reads VDB files, processes the data, and updates GPU resources accordingly.

3. *Rendering:* The render method updates the window surface. It triggers compute shaders for voxel data processing, manages texture and buffer updates, and executes the render pipeline. Additionally, it manages shader hot-reloading, renders the developer GUI and handles screen capture for recording.

10.5 Graphics Pipeline

This section provides an overview of the graphics pipeline initiated at a `RedrawRequest` event.

When the `WgpuContext`'s render method is invoked, it starts by obtaining a reference to the output texture and creates a corresponding view. Following this, a command encoder is initialised to record GPU commands.
Next, it uses the `FrameDescriptor`, a structure designed to transform scene information (including the model, camera, and engine state) stored on the CPU into GPU-compatible bindings. This step prepares all necessary bindings for the compute shaders, executing the ray-tracing algorithm across distributed workgroups, with the results written to a texture.
Once the computation is complete, the rendered image's texture is prepared for display. This process involves creating a vertex shader to generate a full-screen rectangle, onto which the texture is rasterised using fragment shaders, effectively transferring the rendered image to the output texture.
The final phase involves adding the GUI layer over the rendered scene before presenting the completed output texture on the screen.



10.6 GPU Types

This section covers the `FrameDescriptor` data structure and how it generates GPU bindings from the data in `Scene`, which is stored on the CPU.

Virtually the entire ray-tracing algorithm is run in compute shaders, meaning all the information about the model, camera, lights, and metadata must be passed through to the GPU.

The statically sized data, i.e. the camera, sunlight and metadata, is passed in a uniform buffer. This buffer is assembled inside the `FrameDescriptor`, which wraps `ComputeState`.

```
#[repr(C)]
pub struct ComputeState {
  view_projection: [[f32; 4]; 4],
  camera_to_world: [[f32; 4]; 4],
  eye: [f32; 4],
  u: [f32; 4],
  mv: [f32; 4],
  wp: [f32; 4],
  render_mode: [u32; 4],
  show_345: [u32; 4],
  sun_dir: [f32; 4],
  sun_color: [f32; 4],
}
```

The GPU's uniform binding system has strict requirements regarding the types and sizes of data that can be passed to shaders. Therefore, information must be packed into memory-aligned bytes. This is facilitated by the #[repr(C)] attribute, which organizes the struct's layout to match that of a C struct. The data also needs to be padded to fit the aligment options, for that reason all fields are 16 bytes, even if they carry less information.

```
impl ComputeState {
  ...
  pub fn build(
    c: &Camera,
    resolution_width: f32,
    render_mode: RenderMode,
    show_grid: [bool; 3],
    sun_dir3: [f32; 3],
    sun_color3: [f32; 3],
    sun_intensity: f32,
    ) -> Self;
}
```

Listing 3. `ComputeState` build method that transforms CPU data into GPU-ready data

The role of `ComputeState` is to translate high-level CPU structures onto these low-level GPU types. In future sections, the function of the structure fields will be thoroughly detailed.

10.7 Camera

This section explains how the 3D ray-casting camera is implemented. In a ray-tracing engine, a camera casts rays from its eye through the middle of the pixels and into the scene.

Fundamentally, the role of the camera is to convert points from world space into screen space. To that end, a view projection matrix can be constructed from the camera's properties (eye, target, FOV, aspect ratio) that take any point in world space and project it onto camera space.

In order to cast a ray in world space from the camera's eye through the middle of the pixel and into the scene, we need to bring the pixel from screen space into world space. This is the inverse operation of projection, and hence, the projection matrix's inverse matrix is the camera-to-world matrix.

$$\boldsymbol{d_s} = \begin{bmatrix} x - \frac{\text{width}}{2} \\ \frac{\text{height}}{2} - y \\ -\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \end{bmatrix}, \text{ C2W} = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \tag{1}$$

Multiplying gives the pixel coordinates in world space

$$\boldsymbol{d_w} = \begin{bmatrix} x - \frac{\text{width}}{2} \\ \frac{\text{height}}{2} - y \\ -\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} = \begin{bmatrix} (x - \frac{\text{width}}{2})u_x + (\frac{\text{height}}{2} - y)v_x - w_x\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \\ (x - \frac{\text{width}}{2})u_y + (\frac{\text{height}}{2} - y)v_y - w_y\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \\ (x - \frac{\text{width}}{2})u_z + (\frac{\text{height}}{2} - y)v_z - w_z\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \end{bmatrix} \tag{2}$$

Which can be re-written by factoring out constant terms into $\boldsymbol{w'}$:

$$\boldsymbol{d_s} = x\boldsymbol{u} + y * (-\boldsymbol{v}) + \boldsymbol{w'} \tag{3}$$

$$\boldsymbol{w'} = -\boldsymbol{u}\frac{\text{width}}{2} + \boldsymbol{v}\frac{\text{height}}{2} - \boldsymbol{w}\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \tag{4}$$

This form of the ray direction equation is very useful since the vectors $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w'}$ can all be computed once per frame, and then the equation is applied in compute shaders per pixel. This method is explained in more detail in this article[14].

In the implementation, the calculation of these constant vectors is the responsibility of the `ComputeState` data structure; the `build` method (lst. 3) takes in a `Camera` specified by its eye, target, FOV and aspect ratio, and computes the view projection matrix, inverts it to get the camera to world matrix, extracts $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$, then uses the screen's resolution to calculate $\boldsymbol{w'}$. It then packs these vectors into 16-byte arrays.

10.8 Shaders

This section explains the role of the three shader stages in the implementation.

Compute Shaders are the first in the pipeline. They are responsible for performing the entire ray-tracing algorithm. The Compute shader distributes computational power to work groups, which can be considered independent units of execution that handle different parts of the calculation in parallel. Each work group is made up of multiple threads that can execute concurrently, significantly speeding up the process by allowing multiple computations to occur simultaneously. The Compute Shader casts rays from the camera eye through the pixels; intersections with the model determine a pixel's colour based on material properties and record these results on a 2D texture.

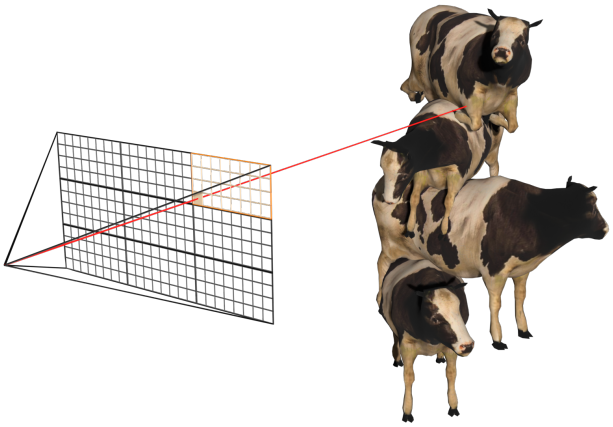


Fig. 3. Compute shader worker casting a camera ray through a pixel. Workgroups of size $8 \times 4 \times 1$ have split up the screen.

Vertex Shaders follow Compute Shaders in the graphics pipeline. Their main role is to define the vertices of a screen-sized rectangle, which serves as the canvas for overlaying the texture computed in the Compute Shader stage.

Fragment Shaders are the last shaders in the pipeline. The Fragment Shaders' role is to rasterize the texture onto the full-screen rectangle prepared by the Vertex Shader. This step effectively transfers the texture onto the display window.

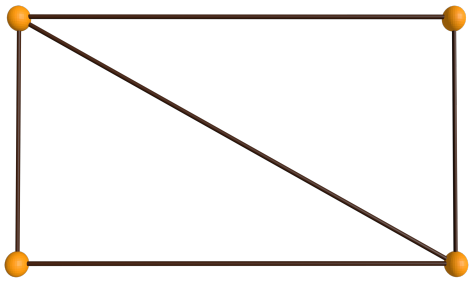


Fig. 4. Vertex shader creating the output surface

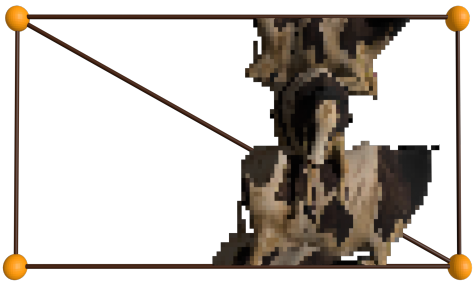


Fig. 5. Fragment shader rasterizing the compute shader texture onto the output surface

10.9 GUI

This section covers the implementation of the GUI that allows the scene to active model to be changed and lighting to be modified, but also provides helpful developer metrics like ms/frame and other benchmarks.

The GUI is managed using the `egui` crate[15]. `egui` is an immediate[16] mode GUI library, which contrasts with traditional retained-mode GUI frameworks[17].

In immediate mode, GUI elements are redrawn every frame and only exist while the code that declares them is running. This approach makes `egui` flexible and responsive, allowing quick updates and changes without needing to manage a complex state or object hierarchy.

The GUI code is run as part of the graphics pipeline in the following steps:

1. *Start Frame:* Each frame begins with a start-up phase where `egui` prepares to receive the definition of the GUI elements. This setup includes handling events from the previous frame, resetting the state as necessary, and preparing to collect new user inputs.

2. *Define GUI Elements:* The application defines its GUI elements by calling functions on an `egui` context object. These functions dynamically create widgets such as buttons, sliders, and text fields based on the current state and user interactions. This step is where the immediate mode shines, as changes to the GUI's state are made directly in response to user actions without requiring a separate update phase.

3. *End Frame:* After all GUI elements are defined, the frame ends with `egui`, rendering all the GUI components onto the screen. During this phase, `egui` computes all elements' final positions and appearances based on interactions and the layout rules provided.

4. *Integration with Graphics Pipeline:* The GUI is overlaid on the application using a texture that `egui` outputs. As in the previous section, this texture is drawn over the application window using a simple full-screen quad.

10.10 Recording

The engine includes an integrated screen recorder designed to efficiently capture screen footage without compromising the frame rate. Unlike external tools such as OBS, which must capture screen output externally and can be slow due to their inability to access application internals, this engine captures the output texture directly before it is displayed on the screen. This method significantly reduces the time required for capture, giving smoother results and keeping the frame rate high.

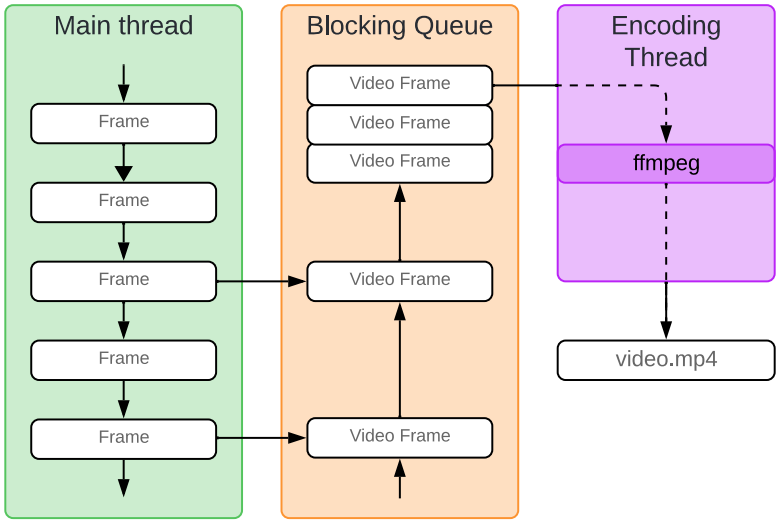


Fig. 6. Producer-Consumer pattern of screen recording implementation

This process's key aspect is ensuring that texture transfer and video encoding are handled asynchronously on a separate thread. This is done using a Producer-Consumer pattern, where the main thread acts as the producer. It periodically places frames into a blocking queue. From this queue, an encoding thread, acting as the consumer, retrieves and processes the frames, encoding them into PNG format and feeding them into `ffmpeg`, a video encoding utility. This approach ensures background processing, minimizing the impact on the engine's performance.

11 VDB Implementation

This section covers the theory and implementation of the VDB data structure.

The VDB (Volumetric Dynamic B-tree) is an advanced data structure designed to efficiently and flexibly represent sparse volumetric data. It is organised hierarchically, consisting of root, internal, and leaf nodes, each serving distinct purposes within the structure. This section begins by explaining in detail how VDB is structured and continues by going through the implementation of the data structure in the rendering engine.

11.1 Data Structure

VDBs are sparse, shallow trees with a fixed depth but expandable breadth capable of covering a virtually infinite spatial domain. This design enables the VDB to efficiently manage large and complex datasets by adjusting the level of detail dynamically and minimising memory usage.

At the heart of the data structure are its three types of nodes: internal root and leaf. The VDB data structure is inherently general; each of the nodes' sizes can be modified depending on the application. However, in practice, only one specialisation of the VDB structure is used: the VDB543. This choice was made because the authors of the original paper[4] analysed a suite of possible shapes and sizes, and this configuration of VDB is the most balanced between performance and memory footprint for most practical applications.

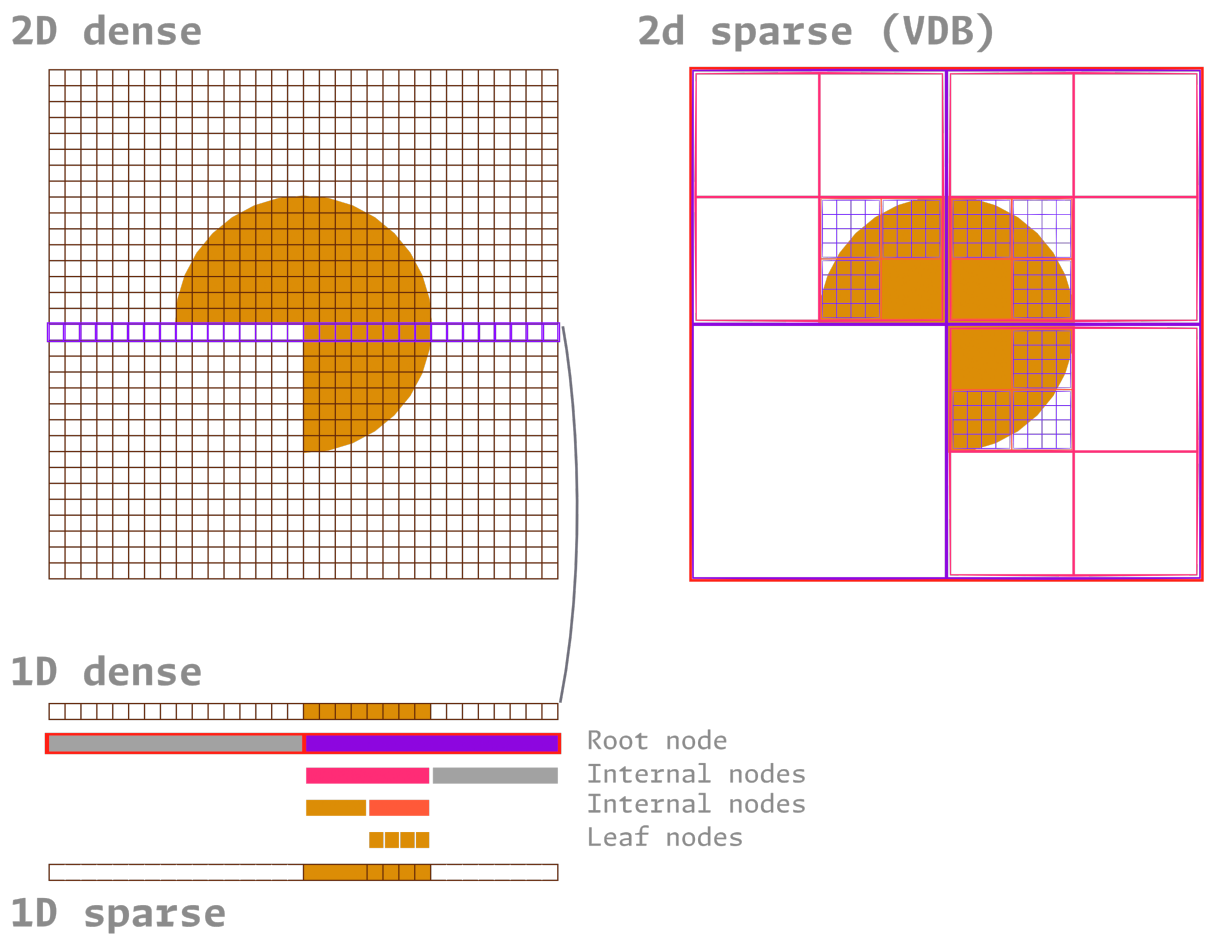


Fig. 7. 2D and 1D slices of the VDB data structure representing three-quarters of a circle. Top left: 2D dense representation of the circle. Top right: 2D sparse representation of the VDB. Bottom left: Sparse representation of the 1D vdb. Usually, VDB nodes have many more child nodes, which would make it harder to visualise; hence, a smaller version of VDB is shown. This figure is an augmented version of the one in the original paper[4]

Leaf Nodes They are the lowest level in the tree structure. They store a 3D cubed grid of side length $2^{\log_2 D}$ (i.e. only powers of 2). A leaf value in the grid can be a voxel's data, other associated data for empty values (such as SDF information), or an empty value. Leaf nodes also store a value mask, a bit array meant to compactly determine whether a value at a specific coordinate in the 3D grid is voxel data or empty.

In the implementation, the trait `Node` is defined, which gives some associated data and methods that leaf and internal nodes have.

```
pub trait Node {
    /// LOG2_D = 3 => '512 = 8 * 8 * 8' values
    const LOG2_D: u64;
    /// Total conceptual LOG2_D node
    const TOTAL_LOG2_D: u64;
    /// Total conceptual LOG2_D of child node
    const CHILD_TOTAL_LOG2_D: u64 = Self::TOTAL_LOG2_D - Self::LOG2_D;
    /// Size of this node (i.e. length of data array)
    const SIZE: usize = 1 << (Self::LOG2_D * 3);
}
```

Listing 4. `Node` trait definition

In lst. 4, `TOTAL_LOG2_D` represents the $\log_2$ of the total dimension of the node, meaning how much actual space the node occupies. Leaf nodes are at the bottom of the tree and do not have children, so this is the same as $\log_2 D$, but this value will be relevant for internal nodes. All other attributes are determined at compile-time depending on the size of the node $\log_2 D$.

> **Sidenote on Coordinate Systems** It is very convenient for side lengths to be powers of two because of the way integers are stored in memory as binary values. To get the global coordinate of a node with `TOTAL_LOG2_D` $= 3$ containing a point in global coordinates, the three least significant bits of each coordinate must be masked out. This operation can be done in a single CPU instruction for each coordinate.
>
> ```
> /// Give global origin of Node coordinates from ‘global‘ point
> fn global_to_node(global: GlobalCoordinates) -> GlobalCoordinates {
>     global.map(|c| (c >> Self::TOTAL_LOG2_D) << Self::TOTAL_LOG2_D)
> }
> ```
>
> Similarly, to get the relative coordinates of a global point within the node, are precisely the `TOTAL_LOG2_D` least significant bits.
>
> ```
> /// Give local coordinates relative to the Node containing ‘global‘ position
> fn global_to_relative(global: GlobalCoordinates) -> LocalCoordinates {
>     global.map(|c| (c & ((1 << Self::TOTAL_LOG2_D) - 1)))
> }
> ```
>
> This pattern of a few bit-wise operations can achieve any conversion between coordinate systems one might need, and all of these through operations are extremely fast to compute on modern CPUs.

Lst. 5 shows a simplified definition of the leaf node data structure in the implementation. It has two fields: data, an array representing the 3D cube grid of values, and a value mask, a bit-mask carrying information on what each value represents, a voxel or empty space. the data array has has $2^{3 \log_2 D}$ entries(e.g. for $\log_2 D = 3 \Rightarrow D = 8$ the leaf node has $8 \times 8 \times 8 = 512 = 2^9$ values). The value mask has the same number of bit entries but is stored as an array of unsigned 64-bit integers. Hence there are $\frac{D^3}{64}$ of them.

```
pub struct LeafNode<ValueType, const LOG2_D: u64>
{
    pub data: [LeafData<ValueType>; (1 << (LOG2_D * 3))],
    pub value_mask: [u64; ((1 << (LOG2_D * 3)) / 64)],
}

pub enum LeafData<ValueType> {
    Tile(usize),
    Value(ValueType),
}

impl<ValueType, const LOG2_D: u64> Node for LeafNode<ValueType, LOG2_D>
{
    const LOG2_D: u64 = LOG2_D;
    const TOTAL_LOG2_D: u64 = LOG2_D;
}
```

Listing 5. `LeafNode` definition. In the original paper[4], node data is set as a union instead of an enum in order to save on memory space, only using the masks to determine the type of a particular value. In this implementation, an enum is used strictly for *ergonomics*, as the extra 1 byte of memory per value is generally not expensive on heap-allocated memory. The value mask will still be crucial for the GPU version of VDB, where effective memory management is more important, and shading languages do not have enum support. In the `Node` trait implementation, since these nodes are the bottom level in the hierarchy (meaning they have no children), their in-memory dimensions are the same as their world space dimensions.

The implementation is general both in the type of value stored at the voxel level, `ValueType`, and in the Node dimension, `LOG2_D`. This is achieved using Rust's generic const expressions feature[18] that is only available on the nightly toolchain. These work in a way akin to C++ templates, allowing the definition of types of static sizes chosen by the data structure user resolved at compile time. This approach allows for customising tree breadth and depth at compile time with no run-time overhead.

Internal Nodes They sit between the root node and the leaf nodes, forming the middle layer of the tree structure. They also store a 3D cubed grid of side length 2^D of values. An internal value can either be a pointer to a child node (leaf or internal) or a tile value, which is a value that is the same for the whole space that a child node in that position would cover. Internal nodes also store a value mask and a child mask. These determine whether a value at a specific coordinate in the 3D grid is a child pointer, value type, or empty value.

```
pub struct InternalNode<ValueType, ChildType: Node, const LOG2_D: u64>
{
    pub data: [InternalData<ChildType>; (1 << (LOG2_D * 3))],
    pub value_mask: [u64; ((1 << (LOG2_D * 3)) / 64)],
    pub child_mask: [u64; ((1 << (LOG2_D * 3)) / 64)],
}

pub enum InternalData<ChildType> {
    Node(Box<ChildType>),
    Tile(u32),
}

impl<ValueType, ChildType: Node, const LOG2_D: u64> Node
    for InternalNode<ValueType, ChildType, LOG2_D>
{
    const LOG2_D: u64 = LOG2_D;
    const TOTAL_LOG2_D: u64 = LOG2_D + ChildType::TOTAL_LOG2_D;
}
```

Listing 6. `InternalNode` definition. Internal nodes have an extra field, the child mask, which is the same size as the value mask. Additionally, the internal data enum now has variants for a child pointer or 4 bytes of memory.

When implementing the `Node` the `TOTAL_LOG2_D` is calculated by adding this node $\log_2 D$ with the child node's total $\log_2 D$. For example, for an internal node with $log_2 D = 4$ with children that are leaf nodes of $log_2 D' = 3$, the internal node's $\log_2 D_{total}$ will be 7. This means that the internal node has $16 \times 16 \times 16$ children that each has $8 \times 8 \times 8$ voxels; the total number of voxels one of these internal nodes is $128 \times 128 \times 128$ or $2^7 \times 2^7 \times 2^7$.

It is important to note that all children of an internal node must be of the same type, which means each level in the tree only has one type of node. This ensures consistency in the coordinate system discussed previously.

Root Node The root node is a single node at the top of the VDB hierarchy. Unlike typical nodes in a tree data structure, the root node in a VDB does not store data directly but instead serves as an entry point to the tree. It contains a hash map indexed by global coordinates, linking to all its child nodes. This setup allows for quick access and updates, as the root node acts as a guide to more detailed data stored deeper in the hierarchy. Because its children nodes are stored by a hash map, it only stores information about space that has information to be stored(unlike an octree, where empty top-level nodes are frequent). The root node's primary role is to organise and provide access to internal nodes.

```
pub struct RootNode<ValueType, ChildType: Node>
{
    pub map: HashMap<GlobalCoordinates, RootData<ChildType>>,
}

pub enum RootData<ChildType> {
    Node(Box<ChildType>),
    Tile(u32),
}
```

Listing 7. `RootNode` definition. `RootData` is either a pointer to a child or 4 bytes of data for a tile value.

Finally, a VDB consists of a root node and some metadata associated with the volume, stored in the `grid_descriptor` field. This metadata is generally only important when reading and writing `.vdb` files.

```
pub struct VDB<ValueType, ChildType: Node>
{
    pub root: RootNode<ValueType, ChildType>,
    pub grid_descriptor: GridDescriptor,
}
```

Listing 8. `VDB` definition

11.2 VDB543

VDB543 is the most widely used configuration of the VDB data structure because it balances performance and memory footprint for most applications.

To refer to different shapes of the VDB data structure, by convention, they are named as $\mathrm{VDB}[\mathrm{a}_0, \mathrm{a}_1, \ldots, \mathrm{a}_\mathrm{n}]$, n layers of internal nodes with $\log_2 D_i = a_i$ followed by a layer of leaf nodes with $\log_2 D_n = a_n$. VDB543 therefore has a layer of leaf nodes with $\log_2 D_{n3} = 3$ and two layers of internal nodes, one with $\log_2 D_{n4} = 4$ and the other with $\log_2 D_{n5} = 5$.

To implement this type of VDB, a new type name for each type of node is created as shown in lst. 9, chaining them up the tree. This section will refer to these nodes as `Node3`s, `Node4`s and `Node5`s, respectively.

```
pub type N3<ValueType> = LeafNode<ValueType, 3>;
pub type N4<ValueType> = InternalNode<ValueType, N3<ValueType>, 4>;
pub type N5<ValueType> = InternalNode<ValueType, N4<ValueType>, 5>;
pub type VDB543<ValueType> = VDB<ValueType, N5<ValueType>>;
```

Listing 9. `VDB543` definition

To calculate how much the in-memory size, in bytes, of each node, the following calculation can be done by taking into account the size of the 3D grid together with the masks:

For leaf nodes: $M = D^3(v+1) + \frac{D^3}{8}$

For internal nodes (2 masks): $M = D^3(v+1) + 2\frac{D^3}{8}$

Where:

$D =$ dimension of node (side-length)

$v =$ number bytes the value type occupies (min. of 4)

Simillarly to find out how many voxels each node covers in world space:

Node3:
$$D = 2^3 = 8$$
$$S = D^3 = 8 \times 8 \times 8 = 512$$

Node4:
$$D = 2^4 = 16$$
$$D_t = 2^{4+3} = 128$$
$$S = D_t^3 = 128 \times 128 \times 128 = 2,097,152$$

Node5:
$$D = 2^5 = 32$$
$$D_t = 2^{5+4+3} = 4096$$
$$S = D_t^3 = 4096 \times 4096 \times 4096 = 68,719,476,736$$

A single Node5 represents 4069^3 voxels in space, just under 69 billion. Here, the power of the VDB data structure can be seen; models can have multiple `Node5`s covering trillions of voxels in total, all of which can be accessed in $O(1)$ time by going three layers down the tree.

11.3 Reading `.vdb files`

VDB was introduced along with an associated file format `.vdb`, which gives a compact data structure representation. This section covers the part of the implementation that reads VDB files and stores them in memory.

Unfortunately, no official documentation of the format used to encode `exists.vdb` files. The only "official" resource available is the OpenVDB codebase[7]. This article[19] reviews a file format reverse engineered from the OpenVDB codebase and the bytecode of `.vdb` files. The implementation uses this latter, reverse engineered format, which only supports `.vdb` from version 218 onwards.

The following is an overview of the contents of a `.vdb` file, as described in [19].

1. Header
   (a) Magic number spelling out " BDV", including the space character (8 bytes)
   (b) File version (u32)
   (c) Library major and minor versions (2 u32s)
   (d) Grid offsets to follow flag (u8)
   (e) UUID (128-bit)
   (f) Metadata entries, length-based list
   (g) Number of grids (u32)

   Since multiple grids can be stored in a single file, the following steps are repeated for all grids.
2. VDB Grid Grids are composed of a grid metadata and a tree. In the engine's implementation, they are just called `VDB`, as seen in lst. 8.
   (a) Name of the grid (length-based string)
   (b) Grid type (length-based string)
   e.g. `Tree_float_5_4_3` refers to a VDB543f32 in the implementation. This type T will determine the data size when reading the tree data later on.
   (c) Instance parent (u32)

(d) Byte offset to grid descriptor (u64)

(e) Start and end location of grid data (2 u64)

(f) Compression flags (u32)

(g) Grid metadata, length-based list of metadata entries
An entry consists of an entry name as a length-based string, an entry type as a length-based list, and actual data

(h) Transform matrices to apply to voxels coordinates to convert from index space to world space (4x4 f64s)

3. VDB Tree `.vdb` files can have multiple trees associated with the grid; hence, this step can be repeated for all trees in a grid.

   (a) the number 1 (u32)

   (b) background value of root node (T)

   (c) Number of tile values in root node (u32)

   (d) Number of children of the root node (u32)

   (e) Descend the tree depth-first describing its *topology*. For each internal node, starting from the top layer. The idea is to first describe the tree topology and the nodes' hierarchy, then in a second pass in item 3f to give the actual voxel data.

      i. The origin of the node, only for top-level nodes (three i32)
      ii. The child mask, only for internal nodes (D^3 bits)
      iii. The value mask (D^3 bits)
      iv. Compression flags
      v. Values, only for internal nodes ($D^3 \times T$)
      vi. Repeat item 3e for every child in order of the mask

   (f) Leaf Node data as value mask + values (D^3 bits + D^3)
   The LeafNodes are given in the same order they were covered in item 3e

In the implementation of VDB file handling within the rendering engine, a specialised streaming reader, termed `VdbReader`, is designed to manage the reading process efficiently. This approach optimises memory usage by incrementally processing the file's content rather than loading the entire file into memory simultaneously. This method is particularly beneficial since VDB files can have hundreds of megabytes.

The `VdbReader` is structured to sequentially process sections of the VDB file, creating nodes as it reads and assembles them into the complete VDB data structure shown in section 11.1.

`VdbReader` reads from the VDB file according to the grid descriptors, constructing nodes from the data, ensuring each node is placed within the VDB hierarchy. This process involves interpreting the file's byte stream according to the VDB format specifications and converting this data into a usable form within the rendering engine.

```
pub struct VdbReader<R: Read + Seek> {
  reader: R,
  pub grid_descriptors: HashMap<String, GridDescriptor>,
}
```

```rust
impl<R: Read + Seek> VdbReader<R> {
  ...
  pub fn read_vdb_grid<T: VdbValueType>(&mut self, name: &str) -> Result<VDB<T>> {
    let grid_descriptor = self.grid_descriptors.get(name).cloned()
    .ok_or_else(|| ErrorKind::InvalidGridName(name.to_owned()))?;
    grid_descriptor.seek_to_grid(&mut self.reader)?;

    if self.header.file_version >= OPENVDB_FILE_VERSION_NODE_MASK_COMPRESSION {
      let _: Compression = self.reader.read_u32::<LittleEndian>()?.try_into()?;
    }
    let _ = Self::read_metadata(&mut self.reader)?;

    let mut vdb = self.read_tree_topology::<T>(&grid_descriptor)?;
    self.read_tree_data::<T>(&grid_descriptor, &mut vdb)?;

    Ok(vdb)
  }
}
```

Listing 10. `VdbReader` definition: `reader` is the file stream handler, `grid_descriptors` hold the metadata given in item 2g. `VdbReader` implementation: The method `read_vdb_grid` is shown, which is called after the file header is handled, and returns a VDB if the file contents match the expectations from the header; if not, it returns an error.

With this reader implemented any `.vdb` file can be brought into the rendering engine, which will provide a series of models from the internet to use during testing and in the results section. Additionally, thanks to the widespread adoption of VDB in modern 3D modelling tools like Blender and Maya, any 3D model can be converted to VDB, essentially enabling any mesh to be pulled into the engine, albeit by stepping out of the engine first.

11.4 GPU VDB

The next step is designing a version of VDB that can be passed to the GPU in compute shaders. The authors of OpenVDB created a version of the VDB data structure, NanoVDB[20], that can use CUDA instructions for GPU acceleration on the data structure. NanoVDB is explained very well by the authors in this article on the Nvidia blog [21].

At this point, this implementation diverges from the OpenVDB library. This project aims to optimise performance while supporting as many platforms as possible, and CUDA instructions are only supported on Nvidia hardware.

This section presents a custom implementation for a GPU-compatible, read-only version of VDB543. This version splits VDBs into two separate components:

(a) Mask group This is a collection of five bindings, each an array of u32s that stores the masks for all nodes: two mask arrays for each internal node layer (Node5 and Node4) and one mask array for the leaf layer (Node3). Each array has length $N\frac{D^3}{32}$, where N is the number of nodes it refers to. These five buffers are passed to the GPU in a storage buffer group, each in a separate binding.

```rust
pub struct MaskUniform {
  kids5: Vec<Node5Mask>,
  vals5: Vec<Node5Mask>,
  kids4: Vec<Node4Mask>,
  vals4: Vec<Node4Mask>,
  vals3: Vec<Node3Mask>,
  origins: Vec<[i32; 4]>,
}
```

```
impl MaskUniform {
  pub fn bind(&self, device: &Device) ->
  ([Buffer; 6], [Vec<u8>; 6], BindGroup, BindGroupLayout) {
    let buffer_contents = self.get_contents();
    let buffers = self.create_buffers(device, &buffer_contents);
    let layout = self.create_bind_group_layout(device);
    let bind_group = self.create_bind_group(&buffers, &layout, device);

    (buffers, buffer_contents, bind_group, layout)
  }
  ...
  fn create_bind_group_layout(&self, device: &Device) -> BindGroupLayout {
    let entries = &(0..=5)
    .map(|binding| wgpu::BindGroupLayoutEntry {
      binding,
      visibility: wgpu::ShaderStages::COMPUTE,
      ty: wgpu::BindingType::Buffer {
        ty: wgpu::BufferBindingType::Storage { read_only: true },
        has_dynamic_offset: false,
        min_binding_size: None,
      },
      count: None,
    }).collect::<Vec<_>>()[..];

    device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
      entries,
      label: Some("Mask␣Bind␣Group␣Layout"),
    })
  }
}
```

Listing 11. `MaskUniform` definition: Each type of mask list is a separate binding. An additional binding is created to store the origins of the top-level Node5 nodes. `MaskUniform` implementation: The `bind` method generates all data needed to pass the mask group to compute shaders. The `create_bind_group_layout` function is the critical part of this process; each binding is a storage buffer type with read-only access.

```
struct Node5Mask { m: array<u32, 1024>, }; // 32^3/32
struct Node4Mask { m: array<u32, 128>, };  // 16^3/32
struct Node3Mask { m: array<u32, 16>, };   // 8^3/32

@group(3) @binding(0)
var<storage, read> kids5: array<Node5Mask>;
@group(3) @binding(1)
var<storage, read> vals5: array<Node5Mask>;
@group(3) @binding(2)
var<storage, read> kids4: array<Node4Mask>;
@group(3) @binding(3)
var<storage, read> vals4: array<Node4Mask>;
@group(3) @binding(4)
var<storage, read> vals3: array<Node3Mask>;
@group(3) @binding(5)
var<storage, read> origins: array<vec3<i32>>;
```

Listing 12. wgsl version of the mask arrays. Each buffer is divided into parts based on the size of the node it tackles. This enables getting all the masks of a node with a particular index by simply indexing them into the mask array. For example, `kids5[0]` would give the first child mask for the Node5 at index 0.

With the bind group created and integrated into the pipeline, compute shaders can now access the topology of the data.

A key part of this representation is the order in which node masks are given in each particular binding, meaning how the node index in the value mask is related to that same node in the CPU-based data structure. The approach is the same as when reading `.vdb` files: a depth-first descent of the tree topology. The order of nodes in the GPU VDB representation is the same as the order in which they appear in the file.

(b) Atlas group This bind group stores all the voxel and child "pointer" packed into atlases as 3D textures. This data type is the perfect storage type for 3D grids because it enables retaining relative coordinates within the nodes while packing them next to each other in memory. This technique was inspired by two articles [22], [23]. The latter presents a packing termed N^3-tree, which presents a method to serialise octree nodes in an array of textures and store indices in that same array to represent children's nodes. This idea is not directly applicable to VDB since nodes at different depths have different sizes, so much of the space would be wasted since most nodes are smaller than the top-level Node5 nodes. However, this idea can be expanded by using a different texture atlas for every type of node and cross-referencing between each layer. For VDB543, This entails having three different atlases, with the values of the first two (corresponding to Node5s and Node4s) indexing into the Node4 and Node3 atlases, respectively.

```
pub struct NodeAtlas {
  size5: [u32; 3],
  size4: [u32; 3],
  size3: [u32; 3],
}

impl NodeAtlas {
  pub fn bind(&self, device: &Device)
  -> ([Texture; 3], BindGroup, BindGroupLayout) {
    let textures = self.create_textures(device);
    let views = textures.iter()
      .map(|texture| self.create_texture_view(&texture))
      .collect::<Vec<_>>().try_into().unwrap();
    let bind_group_layout = self.create_bind_group_layout(device);
    let bind_group = self.create_bind_group(device, &bind_group_layout, &views);

    (textures, bind_group, bind_group_layout)
  }

  fn create_textures(&self, device: &Device) -> [Texture; 3] {
    [self.size5, self.size4, self.size3].map(|[w, h, d]| {
      device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
          width: w, height: h, depth_or_array_layers: d,
        },
        dimension: wgpu::TextureDimension::D3,
        format: wgpu::TextureFormat::R32Uint,
        usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
        ...
      })
    })
  }
  ...
```

```
  fn create_bind_group_layout(&self, device: &Device) -> BindGroupLayout {
    device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor {
      label: Some("Atlas Texture Bind Group Layout"),
      entries: &[0, 1, 2].map(|binding| wgpu::BindGroupLayoutEntry {
        binding,
        visibility: wgpu::ShaderStages::COMPUTE,
        ty: wgpu::BindingType::Texture {
          sample_type: wgpu::TextureSampleType::Uint,
          view_dimension: wgpu::TextureViewDimension::D3,
          multisampled: false,
        },
        count: None,
      }),
    })
  }
}
```

Listing 13. `NodeAtlas` definition: The three fields are the sizes of each atlas. The `create_textures` creates three textures, sets the correct dimensions and sets value type to `R32Uint` represent a 4-byte single channel, which can be used depending on what the value represents. The `create_bind_group_layout` prepare the binding for compute shaders, setting the sample type to `Uint` to ease access to the atlases' data.

With the bindings implemented, fetching these in the shader code is possible, finally bringing VDB data to the GPU.

```
@group(2) @binding(0)
var node5s: texture_3d<u32>;
@group(2) @binding(1)
var node4s: texture_3d<u32>;
@group(2) @binding(2)
var node3s: texture_3d<u32>;
```

Listing 14. wgsl atlas bindings, to get a value in the atlas at point (x, y, z), `textureLoad(x,y,z).r` is called since all the data is stored on the red channel

Since these are 3D textures, a way of converting the indexing in the mask buffers to these texture atlases is needed. Atlases are cube-shaped, and nodes are packed in them by stacking them next to one another with coordinate priority $x > y > z$ in the same depth-first order as discussed previously. This idea yields a straightforward transformation between the indexes of the mask buffers and the coordinates in the texture atlas, shown in lst. 15

```
fn atlas_origin_from_idx(idx: u32, dim: u32) -> vec3<u32> {
  return vec3(idx % dim, (idx / dim) % dim, idx / (dim * dim));
}
```

Listing 15. wgsl transformation from index of node to coordinate in atlas

The two types of VDB were covered: GPU and CPU-based. To convert from the CPU-based structure to the GPU-based structure one simply recursively descends the tree in depth-first order, adding nodes to their corresponding atlases one by one, stacking them in the 3D texture.

11.5 SDF for VDB

This section outlines what distance field data is injected into empty VDB tiles and how this information can be computed from a normal VDB.

In a standard grid, SDFs encode the Manhattan or Chebyshev distance[24] to the nearest voxel. Splitting up the SDF calculation at each level of the hierarchy can extend this to hierarchical grids.

Empty nodes on the higher level of the hierarchy will store the nearest distance to another node that is not empty (meaning it is either a tile value or it is subdivided into children).

Computing the SDF

A generalised (3D) version of the Chamfer Distance Transform algorithm[25] is used to calculate the distance field information. It is essentially a two-pass method.

1. *Forward Pass:* Starting from the top-left corner of the grid, this pass iterates through each cell. It calculates the minimum distance to the nearest feature by considering the already processed neighbours (typically the cells immediately above and to the left of the current cell). Each cell updates the distance by comparing its current value with the already computed values from these neighbours, typically using predefined weights for horizontal/vertical and diagonal steps.

2. *Backward Pass:* Beginning from the bottom-right corner, this pass iterates back through the grid. It now considers the cells that were not accessible in the forward pass (typically the cells immediately below and to the right of the current cell). Again, it updates the distances in each cell by considering the shortest computed distances from these neighbours.

This algorithm has a straightforward 3D generalisation, starting at the top-left-near corner in the first pass and the bottom-right-far corner in the second pass.

This algorithm can be optimised by considering that for hierarchical data structures, SDF data is virtually independent between levels of the hierarchy. This level of independence allows the computation to be run on a separate thread for each layer in the hierarchy.

12 Ray tracing

With the structure of the rendering engine and the primary data structure covered, it is time to bring everything together to construct an image. A ray tracing algorithm is required, first needing an algorithm to cast a ray. This section will cover methods for casting rays and ray-tracing algorithms, as well as how these integrate with the VDB data structure to optimise performance.

12.1 Casting a ray

The background section glossed over a suite of ray-casting algorithms. This section will detail the implementation of three such algorithms, all built on top of each other: DDA, HDDA, and HDDA+SDF.

12.1.1 DDA

DDA works by ray marching from grid intersection to grid intersection, ensuring each voxel is only polled once. The starting values are a source point and a normalised direction vector.

1. Initial Position (ipos): The starting position of the ray within the voxel grid, calculated by flooring the source vector. It gives the indices of the voxel grid where the ray starts.

2. Delta Distance (deltaDist): This calculates how far the ray must travel in each axis to cross a voxel boundary. It is computed by dividing the length of the direction vector (dir) by each component of the direction vector, giving the distance the ray travels in each axis per step.

3. Step (step): A vector determining the direction to step through the grid in each axis (i.e., whether to increment or decrement the index in each dimension). It is determined by the sign of the direction vector components.

4. Side Distance (sideDist): This vector stores the distance the ray needs to travel to hit the next side of a voxel. It starts with the distance to the first boundary from the source position, adjusted by half a voxel size in the direction of travel to ensure the calculation centres within a voxel.

The core idea of the algorithm is that it precomputes the amount one unit of movement on either axis causes the ray to progress (`deltaDist`). It then maintains a record of how far the ray has progressed on each axis and selects the one for which the ray has moved the least, i.e. the closest boundary intersection.

```
const MAX_RAY_STEPS: i32 = 64;
fn cast_ray_dda(src: vec3<f32>, dir: vec3<f32>) -> vec3<f32> {
  var ipos = vec3<i32>(floor(src));
  let deltaDist = abs(vec3<f32>(length(dir)) / dir);
  let step = vec3<i32>(sign(dir));
  var sideDist = (sign(dir) * (vec3<f32>(ipos) - src) + (sign(dir) * 0.5) + 0.5)
    * deltaDist;
  var mask = vec3<bool>(false);

  for (var i: i32 = 0; i < MAX_RAY_STEPS; i++) {
    let val = getVoxel(ipos);
    if (val.hit) {
      return val.color + dot(vec3<f32>(mask) * vec3(0.01, 0.02, 0.03), vec3(1.0));
    }

    var b1 = sideDist.xyz <= sideDist.yzx;
    var b2 = sideDist.xyz <= sideDist.zxy;
    mask = b1 & b2;

    sideDist += vec3<f32>(mask) * deltaDist;
    ipos += vec3<i32>(mask) * step;
  }

  return vec3<f32>(dir);
}
```

Listing 16. `DDA` algorithm

Specifically, at each iteration of the ray marching loop, three operations are performed:

1. Voxel Check: At each step, the algorithm checks whether the current voxel (ipos) is occupied by using the getVoxel(ipos) function. If this function returns true, indicating a hit, the loop exits as the ray has intersected a voxel, and its colour, adjusted to distinguish between the face on which the ray hit, is returned.

2. Boundary Calculation: Determines which voxel boundary (x, y, or z) will be hit next by comparing the distances in sideDist. The logic here involves comparing each component of sideDist against the others to find the smallest value. This is computed using vector comparisons b1 and b2, which are boolean vectors.

3. Update sideDist and ipos: Updates the current side distance and position based on which boundary will be hit next. For example, If the smallest distance is in the x-direction, then the x-component of sideDist and ipos are updated.

12.1.2 HDDA

This section covers the Hierarchical DDA specifically tailored for traversing volumetric data represented in a VDB, using the data structure at last. This method is much more effective in handling large and sparse volumetric datasets due to its hierarchical traversal mechanism, which significantly enhances computational efficiency and scalability.

The key idea is to scale the ray step at each iteration by the lowest level of detail at that point. For example, at a point in an empty Node4, there is no reason not to step directly the side length of a Node4 along an axis since it is guaranteed that there is no voxel data in that node.

The following is a breakdown of the algorithm in lst. 17:

Initialisation

1. The ray's initial position `p` is set to the source point src.
2. The `step` vector determines the traversal direction in the grid is the non-zero sign of the ray.
3. The `step01` vector is used as a flag to determine whether the size on a given axis should be added when calculating step candidates. It essentially discriminates between the cases shown in fig. 8.
4. The `idir` vector represents the inverse of the directional vector. It is precomputed to avoid doing element-wise division in the loop's body since multiplication is faster than division on FPUs.
5. The `mask` has the same role as in standard DDA; it will be used to decide which increment produced the smallest step on the ray.

With this initial information, the ray can start marching along the VDB.

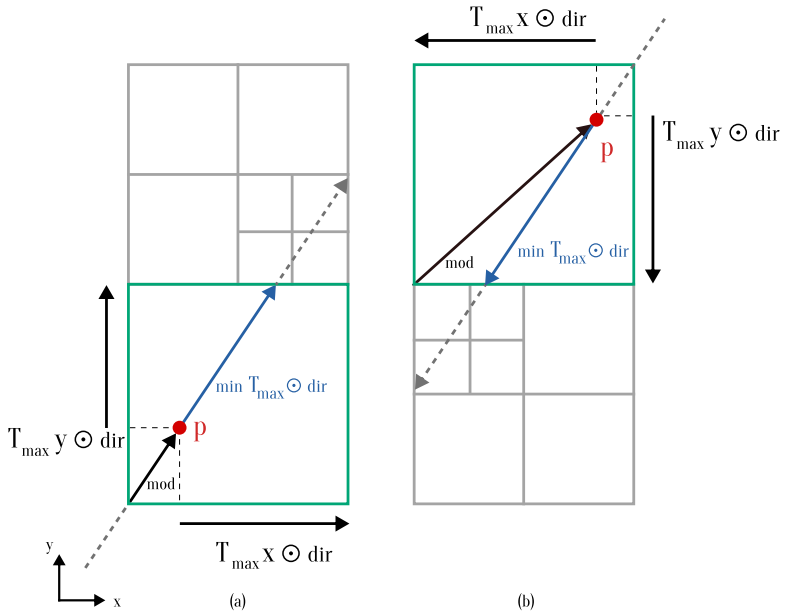


Fig. 8. `HDDA` iteration. (a): Ray positive on both axes. When a ray direction has a positive component, because of the directionality of the modulo operation, it must be subtracted from the size of the node to maintain the ray direction. (b): Ray negative on both axes. When a ray has a negative component, the modulo must simply be inverted since the size is already accounted for.
Blue and grey vectors are candidate steps at the given iteration; each has its components outside the squares. The blue vector has to be selected since it is the shorter one. The black vector represents the modulo-size vector. The ⊙ operation denotes element-wise multiplication. A small branching factor is not typical of the VDB data structure; this was chosen purely for visual clarity.

Loop
The ray traversal loop has the following steps:

1. **Qeuery the VDB** for the value at the current position. If the value is non-empty or out of bounds, return a colour for the voxel
2. **Compute step calculations** scaled by the size of the current level. Compute the candidate steps in all three axes, choose the minimum and step the ray in that direction
3. **Adjust the position** by a tiny factor to ensure it will query the next cell. Since the algorithm deals with points on edges, sampling a point directly on edge is prone to floating point issues, so the position is nudged by a small amount towards the cell in the direction selected at item 2.

```
const HDDA_MAX_RAY_STEPS: u32 = 1000u;
const scale = array<f32, 4>(1., 8., 128., 4096.);
fn hdda_ray(src: vec3<f32>, dir: vec3<f32>) -> vec3<f32> {
  var p: vec3<f32> = src;
  let step: vec3<f32> = sign11(dir);
  let step01: vec3<f32> = max(vec3(0.), step);
  let idir: vec3<f32> = 1. / dir;
  var mask = vec3<bool>();
  var bottom: VdbBottom;

  for(var i: u32 = 0u; i < HDDA_MAX_RAY_STEPS; i++){
    bottom = get_vdb_bot_from_bot(vec3<i32>(floor(p)), bot);

    if !bottom.empty {
      // return voxel color
    }
    if any(p < min_bound || p > max_bound) {
      // return bounding box color
    }
    let size: f32 = scale[3u - bot.num_parents];
    let tMax: vec3<f32> = idir * (size * step01 - modulo_vec3f(p, size));

    p += min(min(tMax.x, tMax.y), tMax.z) * dir;

    let b1 = tMax.xyz <= tMax.yzx;
    let b2 = tMax.xyz <= tMax.zxy;
    mask = b1 & b2;

    p += 4e-4 * step * vec3<f32>(mask);
  }
  // return maximum steps exceeded color
}
```

Listing 17. `HDDA` algorithm

Using the VDB At this point, we can finally make use of the topology of the VDB data structure. The size of the step in HDDA is determined by the lowest node level at that point. This can be computed by stepping down the tree until we get either a voxel value or a node's tile value (a value is also meant to mean empty here). This works fine, but there is some potential for further optimisation.

HDDA usually steps through neighbouring cells, and these neighbouring cells have the same parent (except for cells at the border of the parent's 3D grid). When a ray passes through a Node3, which represents $8 \times 8 \times 8$ voxels, assuming no collision, eight voxel checks are expected, at minimum (when the ray passes parallel to either axis). For each of these eight checks, the lookup in the VDB structure would consist of going from the root node to a Node5 to a Node4, then to a Node3, and then indexing into its grid. However, if the algorithm would remember the chain of nodes from the previous iteration, it could simply check if the voxel it needs is in the Node3 it already has, and use that, if not go up to the parent and check again. These cases are the majority of cases. For eight voxels in a horizontal line, 7 out of 8 would be in this case, all but the first, which would have to go up to the Node4 to get the neighbouring Node3 and then index the voxel. This method optimises lookups for virtually no cost, just using the data structure's topology. This idea would not work in an octree, for example, because in an octree, all nodes are boundary nodes. This is where the grid-like properties of the VDB shine.

In the software implementation, the structure `VdbBottom` (lst. 18) is constructed, meant to represent the path to the bottom of a query in VDB. Along with this data structure, the method `get_vdb_bot_from_bot` (used in lst. 17) is created. The method does exactly what was described above: it tries to get the next value using the nodes of the previous one from the bottom up.

```
struct Parent {
    origin: vec3<i32>,
    idx: u32,
}

struct VdbBottom {
    color: vec3<f32>,
    empty: bool,
    num_parents: u32,
    parents: array<Parent, 3>,
}
```

Listing 18. `VdbBottom` defintion. The structure holds the three (or fewer) parents that the bottom value has, whether the bottom was an empty value, and the colour information if it was voxel.

This HDDA algorithm is already highly efficient (see table 2) and can render complex scenes with high voxel counts and dynamic lighting very well. However, it is always possible to do better.

12.1.3 HDDA+SDF

This section shows the final improvement to the ray casting algorithm. It combines the HDDA algorithm described in the last part with the SDF data described in section 11.5. The idea is simple: at each level of the hierarchy, instead of stepping by one unit in relative space, step by as many as the SDF allows. With the SDF already computed, modifying the HDDA algorithm is as simple as changing a few lines of code and multiplying the step size by the minimum distance.

```
fn hdda(src: vec3<f32>, dir: vec3<f32>) -> vec3<f32> {
  ...
      let size: f32 = scale[3u - bot.num_parents] * bot.dist;
  ...
}
struct VdbBottom {
  ...
  dist: u32, // empty becomes dist
}
```

Listing 19. HDDA+SDF tweaks

The simple HDDA algorithm would have satisfied the goals and objectives set for this project, but this version is even better. At this point, it is time to test these algorithms by adding lights and reflections.

12.2 Sunlight

Sunlight can be described by direction and colour. The idea is to cast camera rays through the scene, and when a ray intersects an object, another ray is cast from that point in the direction of the sun. If that ray intersects anything else, the object is in shadow, so sunlight does not contribute to its colour; otherwise, sunlight hits an object and, therefore, contributes to its colour.

An important part of this algorithm is calculating the contribution of sunlight to an object's colour. A straightforward way would be to add the sunlight colour to the colour of the object times some intensity coefficient. However, the angle at wich the sunlight hits a voxel face is a key aspect of this calculation. The dot product of the sunlight direction and the surface normal of the object can be used to determine how much sunlight falls on the face of an object.

The following model based on the Lambertian Reflectance Formula[26] is proposed:

Ambient Lighting represents indirect light scattered in the environment and illuminates all objects equally.

$$\vec{I}_{ambient} = k_a \cdot C_{ambient} \odot \vec{C}_{surface}$$

Diffuse Lighting represents the light from the sun and affects surfaces based on their orientation relative to the light source.

$$\vec{I}_{diffuse} = k_d \cdot (\vec{L} \cdot \vec{N}) \cdot \vec{C}_{sunlight} \odot \vec{C}_{surface}$$

When an object is **in shadow** it does not receive sunlight, so the $I_{diffuse}$ component is ommited

$$\vec{C}_{total} = \vec{I}_{ambient}$$
$$\vec{C}_{total} = k_a \cdot C_{ambient} \odot \vec{C}_{surface}$$

When an object is **not in shadow** $I_{diffuse}$ is added to the ambient color

$$\vec{C}_{total} = \vec{I}_{ambient} + I_{diffuse}$$
$$\vec{C}_{total} = k_a \cdot C_{ambient} \odot \vec{C}_{surface} + k_d \cdot (\vec{L} \cdot \vec{N}) \cdot \vec{C}_{sunlight} \odot \vec{C}_{surface}$$

Where:

$$k_a = \text{ambient reflectance coeffcient}$$
$$k_d = \text{diffuse reflectance coeffcient}$$
$$\vec{L} = \text{normalized direction vector from the surface to the light source}$$
$$\vec{N} = \text{normalized surface normal}$$
$$\vec{C}_{surface} = \text{color of the surface}$$
$$\vec{C}_{ambient} = \text{ambient color}$$
$$\vec{C}_{snlight} = \text{sunlight color}$$

This model has a straightforward software implementation. The function `hdda_ray` now returns a custom output type that carries information about the result of the ray cast operation, such as state

(hit, miss or out of steps), the render mode, which will be shown in the experiments section fig. 11, the HDDA mask used to determine the surface normal of the intersected voxel, and the point of intersection. The only difference from the mathematical model above is that the alpha channel of the sun's colour is used to encode sunlight strength.

```
fn ray_trace(src: vec3<f32>, dir: vec3<f32>) -> vec3<f32> {
  let hit: HDDAout = hdda_ray(src, dir);
  let step: vec3<f32> = sign11(dir);
  ...
  if hit.state == 0u {
    switch hit.render_mode {
    ...
    case 3u: {
      let N = normalize(-step * vec3<f32>(hit.mask));
      let LN = max(0.0, s.sun_color.a * dot(-s.sun_dir, N));
      let I_d = k_d * s.sun_color.xyz * hit.color * LN;
      let I_a = k_a * AMBIENT_COLOR * hit.color;

      if LN != 0.0  &&
      hdda_ray(hit.p - 4e-2 * step * vec3<f32>(hit.mask), -s.sun_dir).state == 0u {
        return I_a;
      }

      return I_a + I_d;
    }
  }
  ...
}
```

Listing 20. Sunlight rendering on diffuse materials

12.3 Glossy Materials

Adding glossy materials requires bouncing the rays off objects recursively. Unlike perfectly diffuse surfaces that scatter light in all directions, glossy surfaces cause more directed reflections, often leading to visible specular highlights and clear reflections of the environment.

A more complex approach to ray tracing is required, a method that includes handling recursive ray bounces to accurately simulate the reflection phenomena. For glossy materials, when a ray intersects the surface, it typically generates at least two additional types of rays:

1. **Shadow Ray:** This ray is cast towards light sources shown in the previous section.
2. **Reflected Ray:** This ray simulates the reflection of light off the surface, calculated based on the angle of incidence and the normal at the point of intersection. The direction of the reflected ray is given by the formula:

$$\vec{r} = \vec{i} - 2\vec{N}(\vec{i} \cdot \vec{N}) \tag{5}$$

While recursive ray tracing provides a high degree of realism, especially for scenes involving glossy materials, its implementation on GPUs encounters significant challenges. GPUs are not well-suited to handle recursive operations due to their parallel processing architecture. Recursion requires a stack-based memory model, which is more naturally handled by CPUs. Moreover, recursive ray tracing is computationally intensive, especially with multiple bounces (reflections). Each additional bounce increases the complexity exponentially, making real-time rendering particularly challenging.

Nonetheless, "recursive" ray tracing with a very shallow depth of 2 bounces was implemented. This already brings the maximum numbers of rays that can be casted per pixel to 6. This will serve as more of a stress test of the engine and ray-casting algorithms.

The software implementation consists of chaining the same function, copied repeatedly but with different names and setting them up to call each other in sequence. The `ray_trace` function can call the `ray_trace1` function, which can call the `ray_trace2`, all of which can call shadow ray functions.

The colour model of this for glossy materials develops on the previous one, computing the diffuse part of the colour the same way but mixing it with the reflected colour based on a reflectance coefficient k_r The final colour returned by the function is a mix of the ambient plus diffuse lighting and the colour from the reflected ray, weighted by the reflectivity. This mixing accounts for both direct illumination and the effects of reflections from other surfaces.

```
let N = normalize(-step * vec3<f32>(hit.mask));
let rdir = normalize(dir - 2.0 * N * dot(dir, N));
let rsrc = hit.p - 4e-2 * step * vec3<f32>(hit.mask);
let rcol = ray_trace1(rsrc, rdir);
let LN = max(0.0, s.sun_color.a * dot(-s.sun_dir, N));
let I_d = k_d * s.sun_color.xyz * hit.color * LN;
let I_a = k_a * AMBIENT_COLOR * hit.color;

if I != 0.0  &&
hdda_ray(hit.p - 4e-2 * step * vec3<f32>(hit.mask), -s.sun_dir).state == 0u {
  return mix(I_a, rcol, k_r);
}
return mix(I_a + I_d, rcol, k_r);
```

Listing 21. Glossy materials color model

This concludes the methodology section.

Part IV

Results and Experiments

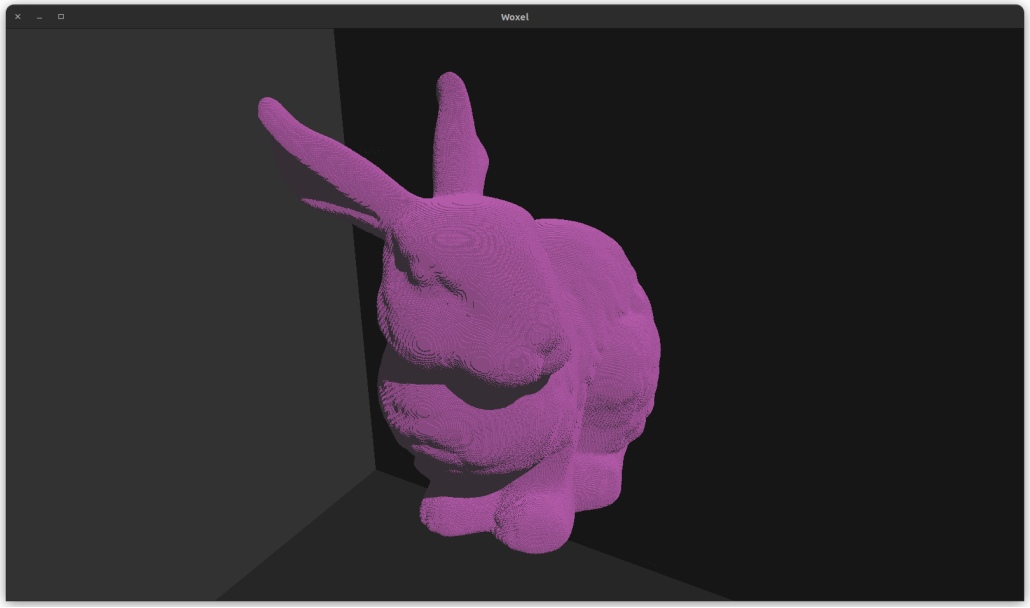


Fig. 9. Bunny with diffuse pink material, model voxel resolution: $628 \times 621 \times 489$

13 Images

This section shows some images captured in the rendering engine. All the models used are samples from the OpenVDB website[27]. Each of the figures below showcases the engine's different functionalities.

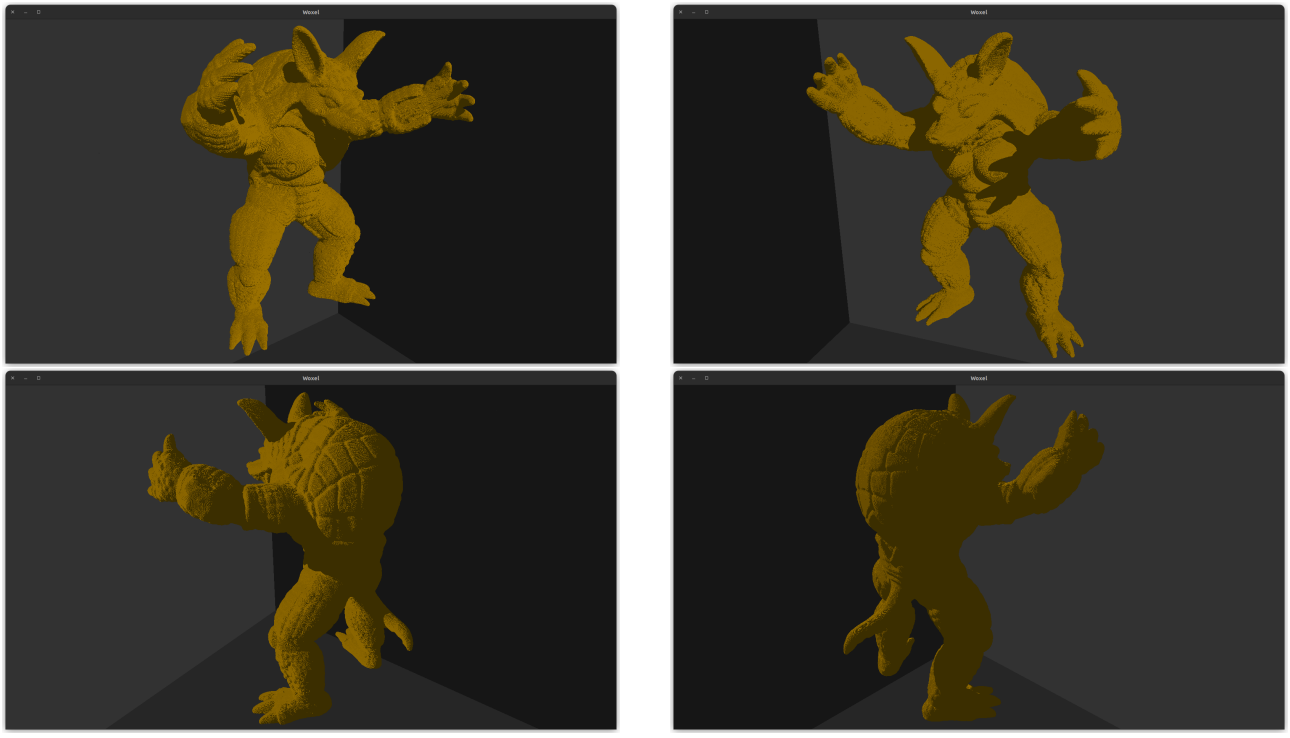
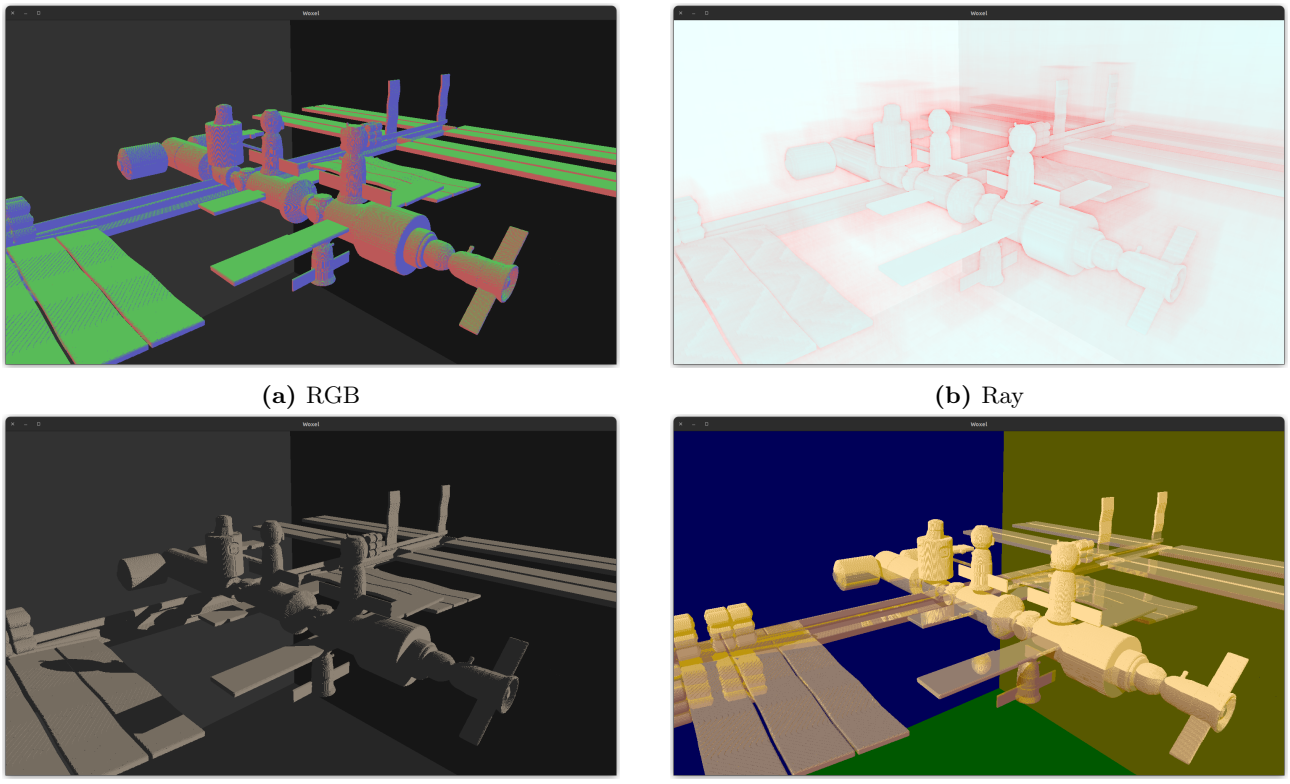


Fig. 10. Multiple angles of an armadillo model with a diffuse material. The voxel resolution of the model is $1276 \times 1518 \times 116$



(a) RGB **(b)** Ray

(c) Diffuse **(d)** Glossy

Fig. 11. Render modes on a ISS model with voxel resolution $4561 \times 617 \times 2999$ (a): RGB mode colors each face based on the axis to which it is parallel. (b): Ray mode colours each pixel based on how many steps the ray took. The colour is interpolated between light blue and red based on how many steps the ray took to go out of bounds or intersect a voxel. The maximum (red) is 200 steps. (c): Diffuse mode shows an object with diffuse material lit by sunlight. (d): Glossy mode shows a half glossy (bottom), half diffuse (top) model lit by sunlight. The out-of-bounds box is coloured to discriminate which face is reflected on which surface. The reflection of the middle pod with a sphere on top can be seen in the central solar panel. More reflections can be seen in the solar panels on the left and right.

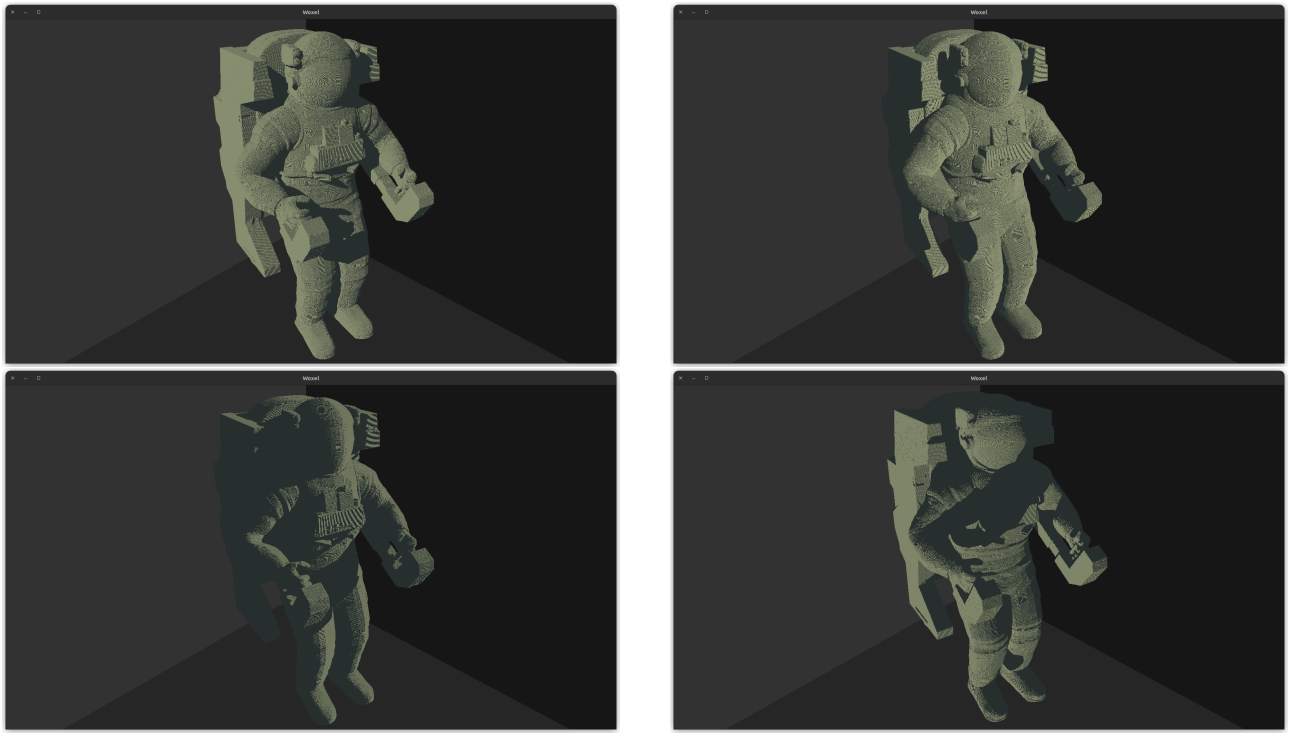


Fig. 12. Dynamic Lighting on an astronaut model. The sunlight is dynamic; its direction, colour and intensity can be changed through the developer GUI in real time. The voxel resolution of the model is $1481 \times 2609 \times 1843$

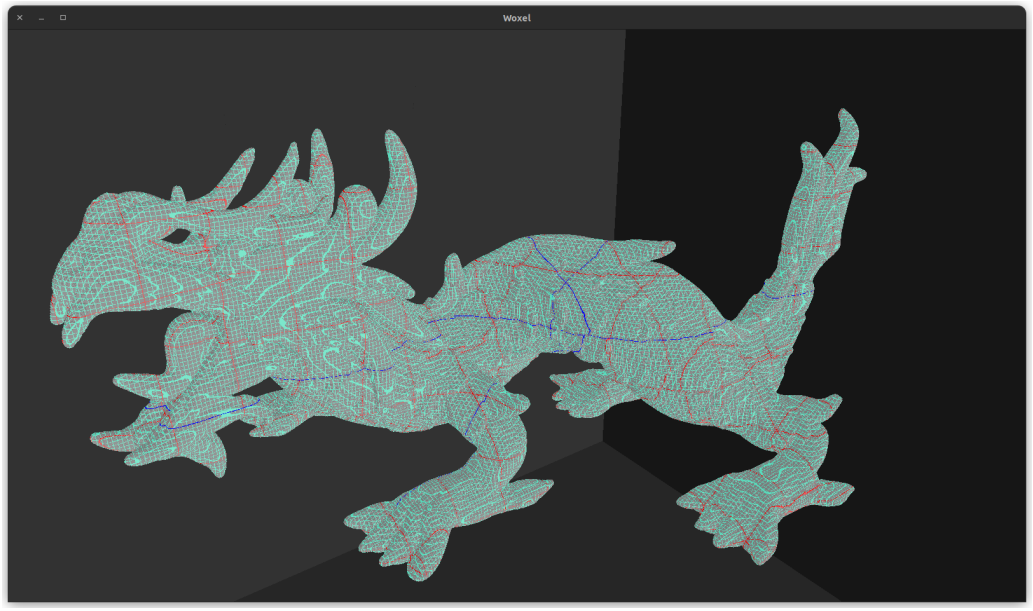


Fig. 13. VDB highligthing: Voxels at the boundaries of VDB nodes are highlighted on a dragon model. The grid structure of the VDB can be seen at each level in the hierarchy. Node3, Node4 and Node5 boundaries are shown in Cyan, Red and Blue, respectively. The voxel resolution of the model is $2023 \times 911 \times 1347$

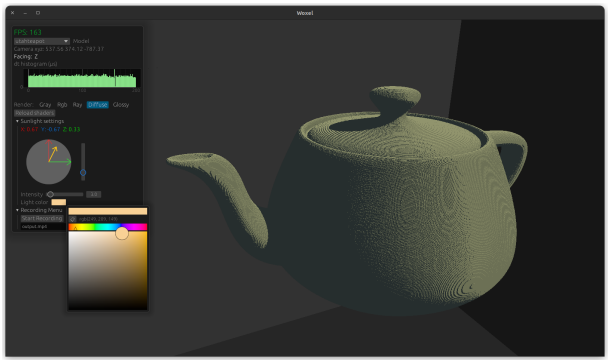


Fig. 14. Developer GUI in engine

The developer GUI has the following uses:

1. Display current FPS and histogram of milliseconds per frame.
2. Changing the model in the viewport through a drop-down menu that scans the assets folder for available models.
3. Camera coordinates and facing direction
4. Functionality to change between the render modes presented in fig. 11
5. The option to reload the shaders while the engine is running.
6. There is a sunlight section available for the diffuse and glossy render modes.
7. The recording menu allows setting an output file and starting or ending the recording.

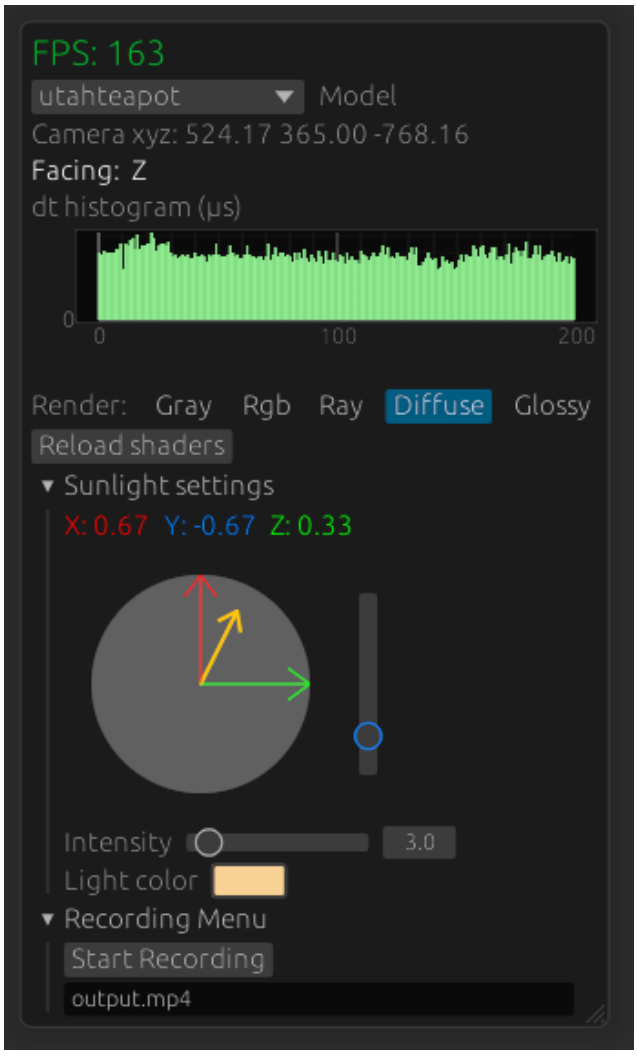


Fig. 15. Developer GUI close-up

14 Experiments

This section compares the ray casting algorithms against each other on different models and render modes.

The specifications of the machine the experiments were run on are presented in table 1.

Experiment machine specifications	
OS	Ubuntu 22.04.3 LTS x86_64
CPU	AMD Ryzen 7 5800H with Radeon
GPU	NVIDIA GeForce RTX 3070 Mobile
RAM	8192MiB
FMA	Enabled

Table 1. Experiment machine specifications

14.1 Comparing DDA, HDDA and HDDA+SDF

The average milliseconds per frame are compared for the DDA, HDDA and HDDA+SDF algorithms on the teapot model (voxel resolution of $981 \times 462 \times 617$) on the ray render mode at three distinct distances from the model, one further away or closer and one near the model. Checking at three separate distances is essential because the performance of the hierarchical algorithms depends on the topology the camera rays are going through. When the target object is far away, most camera rays can traverse the VDB at higher levels in the tree since there is no detail around them. Conversely, when the camera is closer to the object, the rays must traverse more complex topology at lower levels in the tree, and therefore the algorithm can be slower.

	2000m	1000m	500m
DDA	100.3ms*	100.1ms*	50.4ms
HDDA	9.4ms	12.5ms	14.4ms
HDDA+SDF	6ms**	6.4ms	7.6ms

Table 2. Milliseconds per frame of rendering the teapot model using DDA, HDDA, HDDA+SDF at far, medium, and close distances. A voxel is considered 1m × 1m × 1m. The average ms per frame is taken from a 1000 frame interval.
*: The model does not appear in the viewport because the algorithm exceeds the maximum step count of 1000. This time can be treated as the worst-case, casting bouncing the maximum number of times for each pixel.
**: Frame rate cap is hit at 165 FPS; this is as good as the ray casting can get on this machine

(a) **(b)**

Fig. 16. Render in ray mode using (a) HDDA and (b) HDDA+SDF on the teapot model from a 1000m distance. The model resolution is $981 \times 462 \times 617$. As in fig. 11(b), the colour of pixels is determined based on the steps the ray took to intersect the model. The maximum is 200 steps, which corresponds to bright red.

The DDA version cannot handle a mesh of this resolution since rays are marched one voxel at a time. At 2000 voxels away, no ray can reach the object before it exceeds the maximum step count

of 1000. At 500 voxels away, the DDA manages to render part of the teapot at 50 milliseconds per frame (20 FPS), which is not enough for modern engines. Both the HDDA and HDDA+SDF perform very well on this model. The HDDA algorithm has a minimum FPS of 70 at the 500m point. The HDDA+SDF manages to hit the GPU's frame rate cap at 165 FPS at the 2000m point and does not drop below 120 FPS at the close point. Adding the SDF to the HDDA algorithm yields a performance boost of 36% at 2000m, 49% at 1000m, and 47% at 500m.

It is visible in fig. 16 that most of the improvement the SDF offers is for pixels whose corresponding camera rays don't intersect voxels. This is because voxels are intersected relatively quickly in the classic HDDA. The slowest raycasting operations are those that pass voxels very close by and go then off into the background, as shown in fig. 17. This is precisely where the SDF is most valuable since it can get the ray out of those high-detail areas faster. Moreover, when the ray travels in low detail space, there is also an improvement because the further the ray gets from the object, the further it will step. This allows rays to get out of bounds much faster.

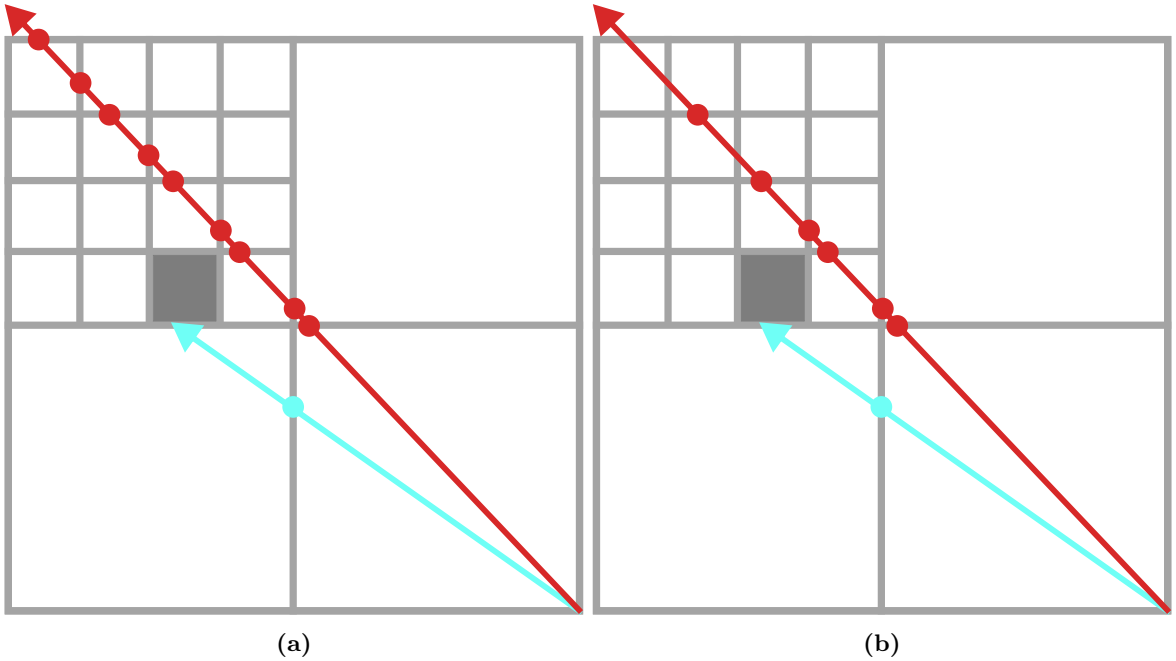


Fig. 17. ray that intersects voxel vs. ray that "near-misses" it. (a) HDDA algorithm; the first has 2 axis crossings, the latter has 9 (b) HDDA+SDF algorithm; the first has 2 axis crossings, the latter has 6.

The same experiment setup is repeated for the ISS model because it has many gaps and tight spaces in its geometry. This time, only HDDA and HDDA+SDF are considered since the DDA could not handle the teapot model, which is much smaller than the ISS. Another change is that the camera positions used are no longer based on the distance to the object but instead positioned such that increasing levels of detail and overlapping geometry are in view.

	pos. 1	pos. 2	pos. 3
HDDA	9.1ms	10.6ms	15.3ms
HDDA+SDF	6ms**	7.2ms	8.3ms
improvement	36%	32%	46%

Table 3. Milliseconds per frame of rendering the ISS model using HDDA, HDDA+SDF at three positions in space selected to be increasingly detailed. The average ms per frame is taken from a 1000 frame interval. The improvement from HDDA to HDDA+SDF is also recorded.
**: Frame rate cap is hit at 165 FPS; this is as good as the ray casting can get on this machine

: (a) (b)

Fig. 18. Render in ray mode using (a) HDDA and (b) HDDA+SDF of the ISS model at position 3. The model resolution is $4561 \times 617 \times 2999$. The colour of pixels is determined analogously to fig. 16. The efficiency of the SDF method in high-detail areas can be clearly seen by comparing the two images: red patches in the middle of the image get much lighter, and bright red areas exist only very close to the model surface.

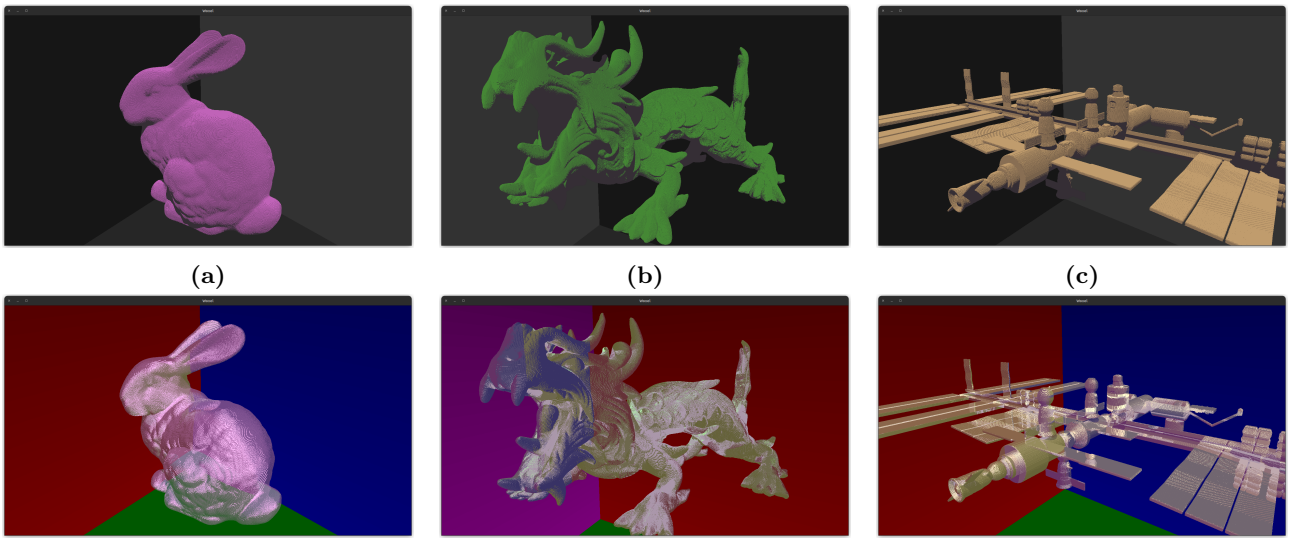
These two experiments prove the efficiency of the HDDA+SDF method; it gives more than 30 % speedup over HDDA in all cases and is able to hit the frame rate cap of 165 FPS in most conditions. The high resolution of the test scenes used also proves that HDDA+SDF is a robust ray-marching algorithm that can handle complex scenes with good performance.

14.2 Performance of HDDA+SDF in ray-tracing

To test the performance of HDDA+SDF when ray-tracing, the average milliseconds per frame will be recorded in a number of scenes for both diffuse and glossy materials.

	bunny	dragon	ISS
diffuse	6ms**	6ms**	6.5ms
glossy	10.2ms	11.6ms	14ms

Table 4. Milliseconds per frame to render each model with a diffuse and a glossy material.
**: Frame rate cap is hit at 165 FPS; this is as good as the ray casting can get on this machine



(a) (b) (c) (d) (e) (f)

Fig. 19. Diffuse (a)-(c) and Glossy (d)-(f) renderings that produce the data int table 4. (a) and (d) shows the bunny model, (b) and (e) shows the dragon model, (c) and (f) show the ISS model

The diffuse rendering scheme performs really well, with a minimum FPS of 150 on the large ISS

model and capping out for the other two. The glossy material has a minimum FPS of 71 on the largest model, with each ray being allowed to be reflected twice. This produces reasonably looking glossy materials, but such a small maximum ray reflection count cannot produce photorealistic results. Scaling up the max ray reflection count is infeasible as the number of rays increases exponentially with it, and two is the maximum the engine can handle while staying above 60 FPS.

This shows that the HDDA+SDF performs very well with diffuse materials and dynamic lighting at over 150 FPS and that it is able to accommodate a small amount of reflections off glossy materials, all with around 80 FPS. This means this algorithm is a viable tool for real-time ray tracing.

Part V
Conclusions

15 Summary of acheivments

Referring to the aims and objectives in sections 2 and 3, the following is a summary of the achievements in this project:

1. The project successfully implemented the Hierarchical Digital Differential Analyzer (HDDA) combined with Signed Distance Fields (SDF) to enhance ray tracing efficiency. This resulted in significant performance gains over traditional HDDA, over 30% faster frame rates.
2. This algorithm was implemented on top of the Volumetric Dynamic B+tree grid acceleration data structure, for which a custom GPU translation was developed.
3. The voxel rendering engine was developed to integrate these advanced algorithms, which proved to handle high-resolution scenes robustly, achieving good performance across tested scenarios, often hitting the framerate cap of 165 FPS.
4. The engine was also equipped with quality-of-life features like benchmarking tooling, recording capabilities, `.vdb` file parsing, hot-reloading of shaders and dynamic lighting editing.
5. The algorithms developed were tested and compared, focusing on their performance in various scenes using different materials and scene complexities. This testing confirmed the algorithms' effectiveness and improved speed and quality of rendering.

16 Challenges and Adaptations

Various challenges were faced in developing this rendering engine, including:

1. **Debugging shaders:** Debugging the wgsl shader code proved particularly challenging due to GPU operations' asynchronous and parallel nature. Traditional debugging tools are not supported on the GPU. Adapting to this meant using external tools like ShaderToy (a browser-based glsl tool) to verify the correctness of shader code segments, which meant frequently translating between wgsl, used by the engine and glsl, supported by ShaderToy.
2. **GPU VDB:** Implementing the VDB data structure on the GPU (GPU VDB) involved overcoming significant technical hurdles related to memory management and data alignment. The adaptation required designing custom data structures and access patterns optimised for GPU memory hierarchies and execution models.
3. **Data Precision:** Maintaining accuracy in GPU calculations, especially for operations prone to floating-point errors, was challenging. Strategies to address this included rearranging the order of operations to minimise rounding errors and precomputing common factors, which helped simplify calculations and preserve the integrity of compounding operations, such as ray marching.

17 Limitations and Future Work

Despite the achievements, several limitations pave the way for future work:

1. **Material Support**: Current material modelling in the engine is somewhat limited. There is no well-rounded implementation of each voxel indexing into a material list; instead, material properties are considered global to the model and are set directly in compute shaders. Future work could add a material list passed into a uniform binding of the shader and into which VDB values index.

2. **Caching and Frustum Culling:** The engine does not yet implement advanced node caching mechanisms or frustum culling, which can significantly improve performance by reducing memory usage. Future versions could integrate these features to handle larger scenes more efficiently.

3. **Loading Multiple Models in the Same Scene:** The current version has no way of loading more than one model at a time into the viewport. Future improvements could focus on adding methods that combine VDBs in order to add more of them to the scene.

18 Reflective Conclusion

This project successfully met its stated aims and objectives, culminating in developing a rendering engine that harnesses modern GPU capabilities for optimised rendering processes on sparse voxel grids. Additionally, a roadmap for future work has been proposed to address and overcome the current limitations identified during the development process.

By implementing this engine, I acquired invaluable insights into the intricate workings of graphics engines and the nuances of GPU profiling. I also gained experience in converting theoretical concepts from scholarly papers to tangible implementations, offering a real-world perspective that bridged academic knowledge with practical application.

Moreover, my technical skills have substantially improved throughout this project. Working extensively with Rust and the wgpu backend, I focused on designing solutions that prioritise performance and scalability. This not only enhanced my programming abilities but also improved my capacity for managing large-scale projects and thoroughly documenting my work.

In addition to its technical achievements, this project contributes to the broader community through its open-source nature, which allows for further development and adaptation by others. By making the tools and findings accessible, the project invites collaboration and continued innovation.

Ultimately, this project not only achieves its initial goals but also lays the groundwork for future advancements. It exemplifies how contemporary technological tools can be effectively utilised to address specific challenges in computer graphics, providing a stepping stone for further exploration and innovation in the field.

References

[1] A. Ducrot and H. Watkins, *An introduction to gks - the graphical kernel system*, eng, Shinfield Park, Reading, 1982. DOI: 10.21957/29uwmqp6a. [Online]. Available: https://www.ecmwf.int/node/9156 (cit. on p. 12).

[2] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1968, 37–45, ISBN: 9781450378970. DOI: 10.1145/1468075.1468082. [Online]. Available: https://doi.org/10.1145/1468075.1468082 (cit. on p. 12).

[3] D. Meagher, *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*, Oct. 1980 (cit. on p. 13).

[4] K. Museth, "Vdb: High-resolution sparse volumes with dynamic topology," *ACM Trans. Graph.*, vol. 32, no. 3, 2013, ISSN: 0730-0301. DOI: 10.1145/2487228.2487235. [Online]. Available: https://doi.org/10.1145/2487228.2487235 (cit. on pp. 13, 24–26, 54).

[5] C. Green, "Improved alpha-tested magnification for vector textures and special effects," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07, San Diego, California: Association for Computing Machinery, 2007, 9–18, ISBN: 9781450318235. DOI: 10.1145/1281500.1281665. [Online]. Available: https://doi.org/10.1145/1281500.1281665 (cit. on p. 15).

[6] K. Museth, "Hierarchical digital differential analyzer for efficient ray-marching in openvdb," in *ACM SIGGRAPH 2014 Talks*, ser. SIGGRAPH '14, Vancouver, Canada: Association for Computing Machinery, 2014, ISBN: 9781450329606. DOI: 10.1145/2614106.2614136. [Online]. Available: https://doi.org/10.1145/2614106.2614136 (cit. on p. 16).

[7] *Openvdb documentation*, Academy Software Foundation(ASWF), 2023. [Online]. Available: https://www.openvdb.org/documentation/doxygen/ (cit. on pp. 16, 29).

[8] *All is cubes*, 2024. [Online]. Available: https://github.com/kpreid/all-is-cubes (cit. on p. 16).

[9] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, ISBN: 1593278284 (cit. on p. 17).

[10] *Wgpu: Rust graphics api*, wgpu Project, 2023. [Online]. Available: https://wgpu.rs/ (cit. on p. 17).

[11] *Webgpu*, World Wide Web Consortium (W3C), 2023. [Online]. Available: https://www.w3.org/TR/webgpu/ (cit. on p. 17).

[12] *Webgpu shading language*, World Wide Web Consortium (W3C), 2024. [Online]. Available: https://www.w3.org/TR/WGSL/ (cit. on p. 17).

[13] *Winit crate*, 2024. [Online]. Available: https://docs.rs/winit/latest/winit/ (cit. on p. 19).

[14] Scratchapixel, *Ray tracing: Generating camera rays*, 2021. [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html (cit. on p. 22).

[15] *Egui crate*, 2024. [Online]. Available: https://docs.rs/egui/latest/egui/ (cit. on p. 23).

[16] C. Muratori, *Immediate-mode graphical user interfaces*, 2005. [Online]. Available: https://caseymuratori.com/blog_0001 (cit. on p. 23).

[17] R. Quinn, *Retained mode versus immediate mode*, 2018. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode (cit. on p. 23).

[18] *The rust reference, 6.14 generic parameters*, 2024. [Online]. Available: https://doc.rust-lang.org/reference/items/generics.html (cit. on p. 27).

[19] S. A. Attrach, *Vdb: A deep dive*, JengaFX, 2022. [Online]. Available: https://jangafx.com/2022/09/29/vdb-a-deep-dive/ (cit. on p. 29).

[20] *Nanovdb documentation*, Academy Software Foundation(ASWF), 2023. [Online]. Available: https://www.openvdb.org/documentation/doxygen/NanoVDB_MainPage.html (cit. on p. 31).

[21] *Accelerating openvdb on gpus with nanovdb*, 2020. [Online]. Available: https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/ (cit. on p. 31).

[22] A. E. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens, "Octree textures on the gpu," in *GPU Gems 2*, M. Pharr, Ed., Accessed: April 2024, Addison-Wesley Professional, 2005, ch. 37. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems2/part-v-image-oriented-computing/chapter-37-octree-textures-gpu (cit. on p. 33).

[23] D. Benson and J. Davis, "Octree textures," *ACM Trans. Graph.*, vol. 21, no. 3, 785–790, 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566652. [Online]. Available: https://doi.org/10.1145/566654.566652 (cit. on p. 33).

[24] C. D. Cantrell, *Modern Mathematical Methods for Physicists and Engineers*. Cambridge University Press, 2000 (cit. on p. 34).

[25] M. Butt and P. Maragos, "Optimum design of chamfer distance transforms," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 7, pp. 1477–84, Feb. 1998. DOI: 10.1109/83.718487 (cit. on p. 35).

[26] S. J. Koppal, "Lambertian reflectance," in *Computer Vision: A Reference Guide*, K. Ikeuchi, Ed. Boston, MA: Springer US, 2014, pp. 441–443, ISBN: 978-0-387-31439-6. DOI: 10.1007/978-0-387-31439-6_534. [Online]. Available: https://doi.org/10.1007/978-0-387-31439-6_534 (cit. on p. 40).

[27] *Openvdb sample models*, Academy Software Foundation(ASWF), 2022. [Online]. Available: https://www.openvdb.org/download/ (cit. on p. 43).

Acronyms

B+tree A m-ary tree with a variable but often large number of children per node.. 12

CUDA Compute Unified Device Architecture. 30

DDA Digital Differential Analyzer, line drawing algorithm described in section 7.3. 14, 53

FOV Field of view, explained in section 10.3, item 3. 20, 21

FPS Frames per second. 18, 47

FPU Floating-Point Unit, a coprocessor for handling floating point numbers. 36

GUI Graphical User Interface. 22

HDDA Hierarchical DDA, line drawing algorithm described in section 7.3. 14

ISS International Space Station. 43, 46

OS Operating System. 17

SDF Signed distance fields, described in section 7.2. 14, 33, 38

UUID Universally Unique Identifier. 28

VDB Volumetric Dynamic B+tree grid data structure introduced by Ken Museth[4]. 11, 18