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: many

Abstract

This is abstract text.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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

Thanks!

1 Introduction

1.1 Motivation

1.2 Aims

1.3 Objectives

1.4 Report structure

2 Background and Literature Review

2.1 Rendering engines

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

The evolution of rendering engines over time reflects the advancments in computational techniques and hardware capabilities enabling more realisitic and immersive experiences

[add history]

2.1.1 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 builiding blocks from which more complex shapes and scenes can be constructed.

Polygons, particullary 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 vertex of a polygon typically associated with additional information such as color, 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 being able to contain color and density information. This charactheristic makes voxels particuallry well-suited for rendering scenes with materials that have intricate internal structures, such as fog, smoke, fire and fluids.

2.1.2 Ray-tracing vs. Rasterization

Rendering engines can utilise two main rendering techniques for rendering scenes: ray tracing and rasterization, both having their advantages and trade-offs.

Rasterization 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 make up polygons, and applying textures and lighting. Over the development of the industry of graphics programming, graphics hardware has become extremely effcient at performing rasterization, making it the standard for video games and interactive applications.

Ray-Tracing, in contrast, simulatesthe 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-quallity images, makeing it favored for applications where visual fidelity is critical. However, recent advancements in hardware have begun to bring real-time ray tracing to interactive applications.

2.2 Representing voxels

To efficiently represent and manipualte voxels in program memory, various data structures can be employed. Each method entails trade-offs between memory usage, access speed and complexity of implementation. Access speed referes to the time complexity of querying the datastructure at an arbitray point in space to retrieve a potential voxel.

2.2.1 Voxel grids

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

However, this simplicity comes at a signficant disatvantage: memory consumption. As the bounding volume or the level of detail of the 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 that are a million units appart in all axes. A voxel grid would have to store all the empty voxels inbetween; 10^{18} memory units reserved, 2 of which carry usefull data. This limitation makes the naive voxel grids impractical for large or highly detailed scenes.

2.2.2 Hierarchical voxel grids (N-trees)

To mitigate this issues, hierarchical grids, such as octrees, are employed. 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 lowere 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 quickly discard large empty or irrelevant regions of space.

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 compared to uniform grids. Another sacrifice is access-time, as querying an arbitrary region of space can entail walking down the tree for several levels. Nonetheless, for applications requiring large, detailed scenes with a mix of dense and sparse regions, the benefits of hierarchical representations often outweigh these drawbacks. This is why N-trees are frequently used in voxel engines.

[add history]

2.2.3 VDB

VDB was introduced in 2013 by Ken Museth[1] 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 can effecitively store tiled data across vast spatial regions.

The VDB data structure was introduced along with a several algorithms that make full use of the data structures features, offering significant improvments in techniques for effciently rendering volumetric data. These are some of the benefits VDB have, as detailed in the orginal 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 benefist make VDB a very compelling data structure to serve as the building block of voxel-based rendering engine.

2.3 Ray tracing

In order to render a scene using ray tracing, camera rays are shot throught the view frustum and into the scene. At each object intersection, part of a ray will get absorbed, reflected and refracted. In order to acheive 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 the integration of ray tracing within the graphics pipeline and the methods used to implement it, focusing on casting a ray through a scene.

2.3.1 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 presented on a screen. While rasterization transforms 3D objects into 2D images through a series of stages(vertex processing, shape assembly, gemoetry shading, rasterization, and fragment processing), the ray tracing pipelines introduces a paradigm shift. It primarily involves calculating the path of rays from the eye (camera) through pixels in a image plane and into the scene, potentially bouncing off surfaces or passing through transparent materials before contributing to the color of a pixel.

This step of calculating a rays path is central in ray tracing, and as such, the performance of the algorithm that does this calculation is critical.

2.3.2 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 will cover methods specific to voxel-based environments.

Ray marching

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

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

While simple, ray marching is not without its 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.

This requires the exploration of more advanced techniques to meet the goal of visual realism and performance.

Ray casting

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

Ray casting finds the intersection of rays with geometric primitives like (e.g. triangles, circles). This method skips stepping along the ray entirley by making use of the underlying mathematics of interseting 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 intersection has linear complexity in the number of polygons in the scene.

SDF

Signed distance fields (SDF) are a different way of representing the enviorment. An SDF provides the minimum distance from a point in space to the closest surface, allowing the ray marching algorithm to efficiently skip empty space and accurately 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.

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

[add history]

2.3.3 Casting a ray on a voxel grid

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

DDA

On a discrete voxel grid, basic ray marching can be improved by stepping from voxel to voxel. Because the voxels are the smallest unit of space, a ray can safely step from one to the next, konwing there is nothing else inbetween.

The Digital Differential Analyzer (DDA) line drawing algorithm does precisley 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 has to travel to intersect a grid line on each axis. At each iteration, it steps to the closest grid intersection along the ray.

[add history]

HDDA

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

[add figure] [add history]

A version of the HDDA algorithm for the VDB data structure was introduced by Ken Museth in 2014[2].

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.

2.4 Summary of simmilar systems

3 Methodology

This section outlines the implementation details of the voxel rendering engine, starting from the selection of programming languages and libraries, going over the architecture of the engine, and diving deep into the data structures and algorithms employed, particularly focusing 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.

3.1 Rust & wgpu

The voxel rendering engine is built using **Rust**, a programming language known for its focus on safety, speed, and concurrency[3]. 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 class 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 encoutered throughout the developmenent of this project.

For the graphical backend, the engine utilizes **wgpu**[4], a Rust library that serves as a safe and portable graphics API. wgpu is designed to run on top of 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[5], aiming for broad support across both native and web platforms. 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 behaviors, common issues in high-performance graphics programming. This 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 throughly.

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 upon which the voxel rendering engine is constructed. 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 the implementation of advanced voxel representation techniques and optimized ray tracing algorithms.

3.2 Engine architecture

The engine's operation is centered 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 it's corresponding handler.

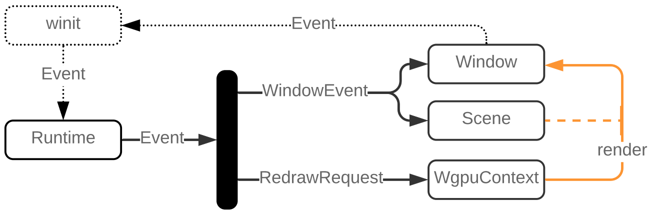


Fig. 1. Engine event-loop diagram. Dotted arrows are implemented in `winit` crate. Black lines represent the flow of events. The arrow line represents the main render function called on the GPU context on the scene for the window.

3.2.1 Runtime

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

- The `Window` is a handler to the engine's graphical window. It is used in filtering OS events that relevant to engine, grabbing the cursor and other boilerplate.
- The `Wgpu Context` holds the creation and application of the rendering pipeline.
- The `Scene` contains information abput the camera and enviorment as well as 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) generaly modify the scene, like the camera position, and therfore are routed to the `Scene` struct.

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

The `RedrawRequested` event is actually emmited in `Runtime`, when it receives the `MainEventsCleared` event, it scheduels the window for a redraw.

3.2.2 Window

The `Window` data structure, included in the **winit**[6] 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 emmits events and the runtime manages and distributes these events accordingly, forming a sort of feedback loop.

3.2.3 Scene

The `Scene` data structe holds information about the enviorment that is 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

In this section, the camera and satte implementation is covered, the model will be covered in later [add link] 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 shouldn't 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, acting as the point from which the scene is observed.
2. *Target:* The point in space the camera is looking at, determining the direction the camera is pointed in.
3. *Field of View (FOV):* An angle representing the range that is in view. In the implementation, this refers to the FOV on the Y (vertical) axis.
4. *Aspect ratio:* The ratio between the width and height of the viewport. It esnures 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. Th 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 the primitives section [add link] where we dive into what information is actually sent to the GPU in compute shaders.

3.2.4 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 follwing responsablities:

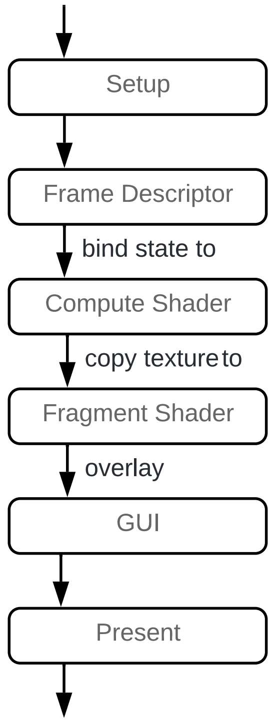
1. *Initialization:* The constructor sets up the wgpu instance, device, queue, and surface. It also configures the surface with the desired format and dimensions, preparing the context for rendering.
2. *Resource Setup:* The constructor prepares various resources such as textures for the atlas representation 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 handles updating 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.

3.2.5 Graphichs Pipeline

This section provides an overview of the graphics pipline that is 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 initialized 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, which then execute the ray-tracing algorithm across distributed workgroups, with the results written to a texture.
Once computation is complete, the texture containing the rendered image is prepared for display. This involves creating a vertex shader to generate a full-screen rectangle, onto which the texture is rasterized 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.



3.2.6 GPU Types

This section covers the `FrameDescriptor` data structe 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. This means all the information about the model, camera, lights, and metadata has to be passed through.

The statically sized data i.e. the camera, sunlight and metadata is passed in an 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. Therfore, 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 structures fields will be detailed thoroughly.

3.2.7 Camera

This section explains how the 3D ray-casting camera is implemented. To role of a camera in a ray-tracing engine is to cast rays from the eye of the camera 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 cameras properties (eye, target, FOV, aspect ratio) that takes any point in world space and projects it onto camera space.

In order to cast a ray in world space from the eye of the camera 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 to projection, and hence the inverse matrix of the projection matrix is the camera-to-world matrix.

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

Multypling 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 constant terms into $\boldsymbol{w'}$:

$$\boldsymbol{d_s} = x\boldsymbol{u} + y * (-\boldsymbol{v}) + \boldsymbol{w'} \quad (3)$$
$$\boldsymbol{w'} = -\boldsymbol{u}\frac{\text{width}}{2} + \boldsymbol{v}\frac{\text{height}}{2} - \boldsymbol{w}\frac{h}{2}\tan^{-1}\frac{\text{fov}}{2} \quad (4)$$

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, then the equation is applied in compute shaders per pixel. This method is explained in more detail in this article[7].

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.

3.2.8 Shaders

In this section the role of the three shader stages in the implementation is explained.

Compute Shaders are the first in the pipeline. They are responsible for performing the entire ray-tracing algorihtm. The Compute shader distributes computational power to work groups, which can be thought as 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 at the same time. The Compute Shader casts rays from the camera eye through the pixels, intersections with the model determine to a pixels's color based on material properties, and record these results on a 2D texture.

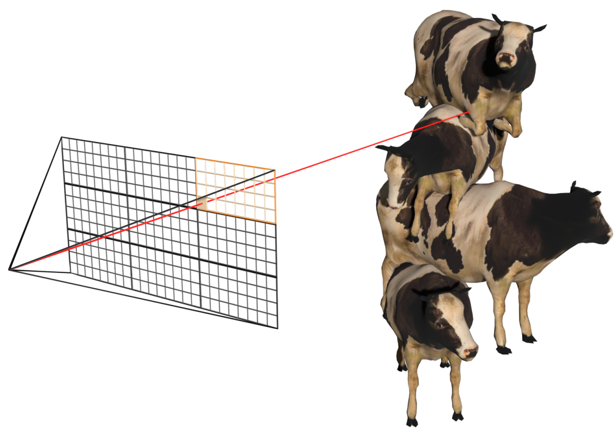


Fig. 2. Compute shader worker casting a camera ray through a pixel. Work groups of size $8 \times 4 \times 1$ have split up the screen.

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.

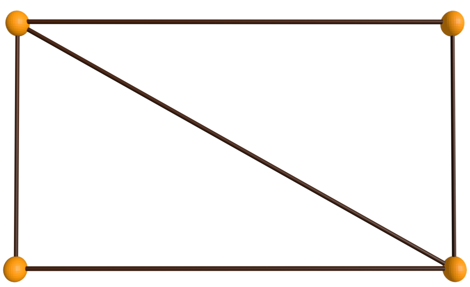


Fig. 3. Vertex shader creating the output surface

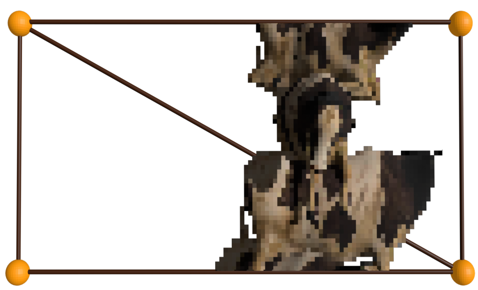


Fig. 4. Fragment shader rasterizing the compute shader texture onto the output surface

3.2.9 GUI

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

The GUI is managed using the `egui` crate[8]. `egui` is an immediate[9] mode GUI library, which contrasts with traditional retained mode GUI frameworks[10].

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, as it allows for 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 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 create widgets such as buttons, sliders, and text fields dynamically, 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 the final positions and appearances of all elements 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. This texture is then drawn over the application window using a simple full-screen quad as in the previous section.

[maybe add screen shot?]

3.2.10 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.

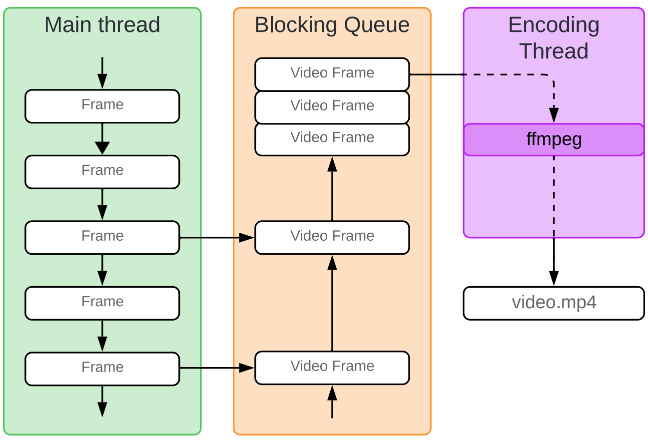


Fig. 5. Producer-Consumer pattern of screen recording implementation

The key aspect of this process is to ensure 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. This includes encoding the frames into PNG format and subsequently feeding them into `ffmpeg`, a video encoding utility. This approach ensures background processing, minimizing the impact on the engine's performance.

3.3 VDB Implementation

In this section the theory and implementation the VDB data structure is covered.

The VDB (Volumetric Dynamic B-tree) is an advanced data structure designed for efficient and flexible representation of sparse volumetric data. It is organized hierarchically, consisting of root nodes, internal nodes, and leaf nodes, each serving distinct purposes within the structure. This section begins by explaining in detail how VDB is structured, and it continues by going though the implementation of the data structure in the rendering engine.

3.3.1 Data Structure

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

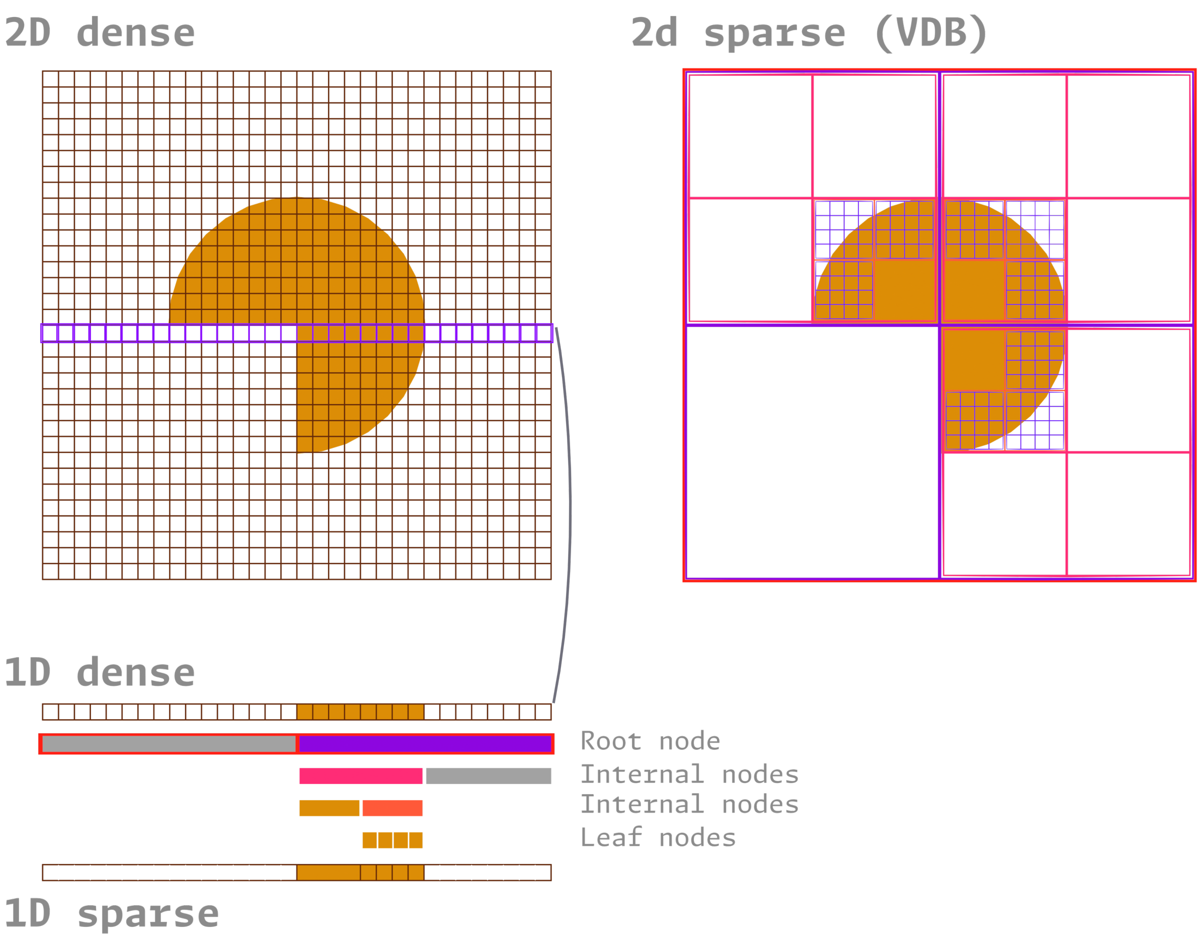


Fig. 6. 2D & 1D slices of the VDB data structure representing three quarters of a circle. Top left: 2D dense representation of the circle. Top right: 2D sparse representation of the VDB. Bottom left: Sparse representation of the 1D vdb. Usually, VDB nodes have many more child nodes, which whould make it harder to visualise, hence a smaller version of VDB is shown. This figure is an augumented version of the one in the original paper[1]

At the hear 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 modefied depending on the application. However, in practice only one specialization of the VDB structure is used, that is the VDB543. This is because the authors of the original paper[1] analyized a suite of possible shapes and sizes, and this configuration of VDB the most balanced between performance and memory footprint for most practical applications [TODO: what applications?]

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). An 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. This is a bit-array meant to compactly determine if value at a specific coordinate in the 3D grid is voxel data or an empty value.

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

```
pub trait Node {
    /// LOG2_D of side length
    /// 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;
    /// Side length
    const DIM: u64 = 1 << Self::LOG2_D;
    /// Total conceptual dimension
    const TOTAL_DIM: u64 = 1 << Self::TOTAL_LOG2_D;
    /// Size of this node (i.e. length of data array)
    const SIZE: usize = 1 << (Self::LOG2_D * 3);
    /// Total conceptual size of node, including child size
    const TOTAL_SIZE: u64 = 1 << (Self::TOTAL_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 don't 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` $= 3$ that contains a point in global coordinates, the 3 least signifcant bits of each coordinate have to be masked out. This can essentially 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)
}
```

Simillary, to get the relative coordinates of a global point within the node are precisely the `TOTAL_LOG2_D` least siginificant 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 acheive any conversion from between coordinate systems one might need, and all of these through operations are extremly 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 which is an array representing the 3D cube grid of values, and value mask which is a the bit-mask carrying information on what each value represnts, 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 it is stored as an array of unsined 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[1] node data is set as a union instead of a enum, in order to save on memory space, only using the masks to determine the type of a particular values. In this implementation a 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 curcial for the GPU version of VDB where there is more need for effective memory management 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 implenetation is general both in the type of value that is stored at the voxel level, `ValueType`, and in the dimension of the Node, `LOG2_D`. This makes use of Rust's generic const expresions feature[11] that is only available on the nightly toolchain. These work in a way akin to C++ templates allowing to define types of static size chosen by the user of the data structure that are resolved at compile time. This approach effectively allows to costumize the 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 would be covered by a child node in that position. Internal nodes also store a value mask and child mask. These determine if value at a specific coordinate in the 3D grid is 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 that is the same size of the value mask. Aditionally the internal data enum now has variants for a child pointer or 4 bytes of memory.

When implemening the `Node` the `TOTAL_LOG2_D` is calculated by adding this nodes $\log_2 D$ 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 have $8 \times 8 \times 8$ 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 imporant 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 ensure 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(unlinke an octree where empty top level nodes are frequent). The root node's primary role is to organize 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 a 4 bytes of data for a tile value.

Finally, a VDB simply consits of a root node and some metadata associated with the volume, stored in the `grid_descriptor` feild. This metadata is generally only imprtant when reading enad writing `.vdb` files.

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

```
    pub grid_descriptor: GridDescriptor,
}
```

Listing 8. `VDB` definition

3.3.2 VDB543

VDB543 is the most widely used configuration of the VDB data structure, because gives a good balance of 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 new type name for each type of node is created as show 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 follwing 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 represent 4069^3 voxels in space, just under 69 billion. This is where 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 acessed in $O(1)$ time, by going three layers down the tree.

3.3.3 Reading `.vdb` files

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

Unfortunately, there is no official documentation of the format used to encode `.vdb` files. The only "official" resource available is the OpenVDB codebase[12]. This article[13] goes over a file format reversed engineered from the OpenVDB codebase and the bytecode of `.vdb` files. The implementation uses this latter, reversed 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 [13].

1. Header

    (a) Magic number spelling out " BDV" (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 size of the data when reading the tree data later on.

    (c) Instance parent (u32)

    (d) Byte offset to grid discriptor (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 consits of entry name as length-based string, entry type as 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 grid, hence after 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 down 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, the hierarchy of nodes, and 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 specialized streaming reader, termed `VdbReader`, is designed to manage the reading process efficiently. This approach optimizes memory usage by processing the file's content incrementally, rather than loading the entire file into memory at once. This method is particularly beneficial since VDB files can have hundres 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 3.3.1.

`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, 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>,
}

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 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 is the file contents matched 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 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.

3.3.4 GPU VDB

The next step is desining 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[14] that can use CUDA instructions to GPU acceleration on the data structure. NanoVDB is explained very well by the authors in this article on the Nvidia blog [15].

At this point this implementation diverges with what the OpenVDB library. The goal of this project is to optimize performance while supporting as many platforms as possible, and CUDA instructions are only supported on Nvidia hardware.

In this section a custom implementation for a GPU compatibale, read-only version of VDB543 is presented. This version splits VDBs into two sperate components:

1. **Mask group** This is a collection of 5 bindings, each an array of u32s that stores the masks for all nodes: 2 mask arrays for each internal node layer (Node5 and Node4) and 1 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 5 buffers are passed two the gpu in a storage buffer group each in a separate binding.

```
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` defintion: Each type of mask list is a separate binding. And additional binding created to store the origins of the top leval 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 crtical part of this process, each binding is a storage buffer type with read-only access.

With the bind group created and integrated in the pipline compute shaders can now acess the topology of the datastructure

```
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 indexinf into the ray. For example, `kids5[0]` would gives the first child mask for the Node5 at index 0.

A key part of this representation is the order in which node masks are given in each particular binding, meaning what how is the index of a Node in the value mask 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. This means the order nodes are in the GPU VDB representation is the same as the order they are given in the file.

2. **Atlas group** This is the bind group that stores all the voxel and child "pointer" in packed into atlases as 3D textures. This data type is the perfect storage type for 3D grids. This gives the ability to retain relative coordinates within the nodes while packing next to each other in memory. This technique was inspired by two articles **octree:1**, **octree:2**. The latter presents a packing termed N^3-tree which presents a method to serialize octree nodes in an array of textures and store indicies in that same array to represent children nodes. This idea isn't directly applicable to VDB since nodes at different depths have different sizes so a lot of space would

be wasted since the majoirty of the nodes are much smaller than the top level Node5 nodes. This can, however, be expanded by using a different texture atlas for every different type of node and cross referencing between each layer. For VDB543 This entails having three different atlases, with the first two's values Node5 and Node4 indexing into the Node4 and Node3 atlases.

```
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,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D3,
        format: wgpu::TextureFormat::R32Uint,
        usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
        label: Some("Atlas Texture"),
        view_formats: &[],
      })
    })
  }
  ...

  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 in are the sizes of each atlas. The `create_textures` creates 3 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` preparse the binding for compute shaders, setting the sample type to `Uint` to give eas of acess to the atlases; data.

With the bindings implemented it is possible to fetch these in the shader code, finally bringing or 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. To that end, atlases are cube shaped, and nodes are packed in it by stacking them next to one another with coorinate priority $x > y > z$ in the same depth-first order as disscused previously. This idea yields a very simple transformation beetween the indexes of the mask bufferes and the coordinates in the texture atlas, shown in **??**

```
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 where covered GPU and CPU-based. To convert from the CPU-based structure to the GPU-based structure one simply recursively descends the tree in deaoth-first order, adding nodes to their corresponding atlases one by one, stacking them in the 3D texture.

3.3.5 Computing SDF

References

[1] 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. 10, 22–24, 32).

[2] 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. 13).

[3] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, ISBN: 1593278284 (cit. on p. 14).

[4] wgpu Project, *Wgpu: Rust graphics api*, 2023. [Online]. Available: https://wgpu.rs/ (cit. on p. 14).

[5] *Webgpu*, World Wide Web Consortium (W3C), 2023. [Online]. Available: https://www.w3.org/TR/webgpu/ (cit. on p. 14).

[6] *Winit crate*, 2024. [Online]. Available: https://docs.rs/winit/latest/winit/ (cit. on p. 15).

[7] 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. 19).

[8] *Egui crate*, 2024. [Online]. Available: https://docs.rs/egui/latest/egui/ (cit. on p. 20).

[9] C. Muratori, *Immediate-mode graphical user interfaces*, 2005. [Online]. Available: https://caseymuratori.com/blog_0001 (cit. on p. 20).

[10] 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. 20).

[11] *The rust reference, 6.14 generic parameters*, 2024. [Online]. Available: https://doc.rust-lang.org/reference/items/generics.html (cit. on p. 24).

[12] *Openvdb documentation*, Academy Software Foundation(ASWF), 2023. [Online]. Available: https://www.openvdb.org/documentation/doxygen/ (cit. on p. 27).

[13] S. A. Attrach, *Vdb: A deep dive*, JengaFX, 2022. [Online]. Available: https://jangafx.com/2022/09/29/vdb-a-deep-dive/ (cit. on p. 27).

[14] *Nanovdb documentation*, Academy Software Foundation(ASWF), 2023. [Online]. Available: https://www.openvdb.org/documentation/doxygen/NanoVDB_MainPage.html (cit. on p. 29).

[15] *Accelerating openvdb on gpus with nanovdb*, 2020. [Online]. Available: https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/ (cit. on p. 29).

Acronyms

B+tree A m-ary tree with a variable but often large number of children per node.. 11

CUDA Compute Unified Device Architecture. 29

DDA Digital Differential Analyzer, line drawing algorithm described in section 2.3.3. 12, 32

FOV Field of view, explained in section 3.2.3, item 3. 18, 19

FPS Frames per second. 16

GUI Graphical User Interface. 20

HDDA Hierarchical DDA, line drawing algorithm described in section 2.3.3. 13

OS Operating System. 15

SDF Signed distance fields, described in section 2.3.2. 12

UUID Universally Unique Identifier. 27

VDB Volumetric Dynamic B+tree grid data structure introduced by Ken Museth[1]. 10, 16

Appendices

A Project outline

Project outline as submitted at the start of the project is a required appendix.

B Risk assessment

Risk assessment is a required appendix. Put here. And there as well