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

What should I aknowledge?! I am citing [1], and [2] hell

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 by Ken Museth[3] 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 primarly 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[4].

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[5]. 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 develpomenent of this project.

For the graphical backend, the engine utilizes **wgpu**[6], 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[7], 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

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

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, roting each event to it's corresponding handler.

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.

3.2.2 WgpuContext

The WgpuContext structure forms the backbone of the rendering pipeline in the voxel rendering engine. It encapsulates the necessary components for interfacing with the GPU using the wgpu API, managing resources such as textures, shaders, and buffers, and executing rendering commands. This structure facilitates the integration of complex rendering techniques, including the handling of volumetric data through VDB and optimized ray casting for voxel environments.

The main components and functionalities of WgpuContext include:

Graphics API Integration: Initializes and configures the wgpu::Device and wgpu::Queue, along with a wgpu::Surface for rendering output. These components are crucial for executing GPU operations and presenting the rendered images. Resource Management: Manages various GPU resources such as Texture, Buffer, and ShaderModule objects. This includes the setup of texture atlases for voxel data, uniform buffers for shader inputs, and dynamic shader loading and compilation. Rendering

Pipeline Configuration: Constructs the rendering pipeline with customized BindGroup and Pipeline configurations tailored to voxel rendering. This setup enables efficient processing and rendering of volumetric data. Event-Driven Rendering Control: Implements methods to handle resizing events and rendering requests, adjusting the viewport and reconfiguring the pipeline as needed. Volumetric Data Handling: Integrates with VDB to load, process, and render volumetric data, including the generation of signed distance fields (SDF) and the management of voxel atlases. GUI Integration: Incorporates EguiDev for GUI rendering over the 3D scene, allowing for interactive model selection and parameter adjustments. Asynchronous and Concurrent Execution: Utilizes tokio::runtime for asynchronous operations and concurrency management, facilitating non-blocking GPU resource operations and efficient data processing. Key implementations derived from the source code:

Initialization and Configuration: The new async function initializes the WgpuContext by setting up the wgpu instance, device, queue, and surface. It also configures the surface with the desired format and dimensions, preparing the context for rendering. 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. Rendering Execution: The render method orchestrates the rendering process. It triggers compute shaders for voxel data processing, manages texture and buffer updates, and executes the render pipeline to draw the scene. Additionally, it integrates GUI rendering and handles screen capture for recording. Shader Management: Provides mechanisms for loading and reloading shaders at runtime, allowing for dynamic updates to the rendering logic without restarting the application. Event Handling and Interaction: Includes functions for responding to window and input events, adjusting the rendering context based on user actions such as resizing the window or changing the viewed model. The WgpuContext architecture demonstrates a comprehensive approach to managing the complexities of rendering voxel-based scenes with ray tracing. It leverages modern GPU programming techniques, efficient data structures, and asynchronous processing to achieve high-performance rendering of volumetric data, while also providing a flexible platform for future enhancements and optimizations.

3.2.3 Graphichs Pipeline

3.2.4 GPU Types

3.2.5 Camera

3.2.6 Shaders

3.2.7 Utilities

3.3 VDB Implementation

References

[1] A. J. Casson and E. Rodriguez-Villegas, "Toward online data reduction for portable electroencephalography systems in epilepsy," *IEEE T. Biomed. Eng.*, vol. 56, no. 12, pp. 2816–2825, 2009 (cit. on p. 6).

[2] A. J. Casson and E. Rodriguez-Villegas, "Toward online data reduction for portable electroencephalography systems in epilepsy," *IEEE T. Biomed. Eng.*, vol. 56, no. 12, pp. 2816–2825, 2009 (cit. on p. 6).

[3] 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. 9, 16).

[4] 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. 12).

[5] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, ISBN: 1593278284 (cit. on p. 13).

[6] wgpu Project, *Wgpu: Rust graphics api*, 2023. [Online]. Available: https://wgpu.rs/ (cit. on p. 13).

[7] *Webgpu*, World Wide Web Consortium (W3C), 2023. [Online]. Available: https://www.w3.org/TR/webgpu/ (cit. on p. 13).

Acronyms

B+tree A m-ary tree with a variable but often large number of children per node.. 10

DDA Digital Differential Analyzer, line drawing algorithm described in section 2.3.3. 11, 16

HDDA Hierarchical DDA, line drawing algorithm described in section 2.3.3. 12

OS Operating System. 14

SDF Signed distance fields, described in section 2.3.2. 11

VDB Volumetric Dynamic B+tree grid data structure introduced by Ken Museth[3]. 9

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