Fabian Huber
#12-135542 / fabian.hbr@protonmail.ch

BACHELOR OF SCIENCE
Games Programming

Voxel rendering using Ray Tracing

Implementation in a Vulkan application using Jai

BACHELOR THESIS - 6GSC0XF101.2
Supervisors: Nicolas Siorak, Elias Farhan

SAE Institute Geneva

July 19th, 2024
13521 Words

I, Fabian Huber, certify having personally written this Bachelor thesis.

I also certify that I have not resorted to plagiarism and have conscientiously and clearly mentioned all borrowings from others.

Geneva, July 19th, 2024

Abstract

Voxel rendering is a technique used in computer graphics to represent three-dimensional objects through volumetric pixels (voxels). This method is particularly useful in applications such as medical imagery, terrain modelling, and game development. With the advent of high-performance graphics APIs like Vulkan, there is potential for significant improvements in the efficiency and quality of voxel rendering. The Jai programming language offers concise syntax and powerful features suited for graphics programming.

This thesis aims to develop an efficient voxel rendering pipeline utilizing ray tracing within a Vulkan-based application. The implementation is carried out leveraging Jai's capabilities for improved development efficiency and performance.

First, key terms such as voxels, ray tracing, ray casting, and path tracing are defined. Two families of voxel rendering techniques - surface extraction and volume marching — are presented. This study involves the integration of ray tracing techniques into a Vulkan application, programmed in Jai. Key aspects include the implementation of primary and secondary ray tracing for accurate light simulation, and voxel traversal algorithms for efficient rendering. Metrics such as frame time, memory consumption, and voxel counts are analysed.

The research confirms that combining voxel rendering with ray tracing in a Vulkan application using Jai can achieve high-performance and high-quality results. These findings highlight the potential for further advancements in real-time graphics applications. Future work will explore additional optimization techniques and the incorporation of more complex lighting models to enhance rendering fidelity.

Keywords: Voxel Rendering, Ray Tracing, Vulkan, Jai Programming Language, Real-Time Graphics, Performance Optimization

Table of Contents

Glossary

API **– Application Programming Interface**: The interface by which two computer programs communicate. 12, 15, 18, 38, 40, 52

automatic exposure: A technique that simulates a camera's automatic exposure adjustments during rendering. It analyses the scene's lighting and adjusts rendered colours to achieve a balanced and natural-looking image within the display's dynamic range. 30

bit mask: A binary pattern used to selectively manipulate individual bits within a binary data word. 17

bloom: A post-processing effect that simulates the visual phenomenon of bright light bleeding into surrounding areas, creating a soft glow. 30

BRDF **– Bidirectional Reflectance Distribution Function**: A function that defines how light is reflected at an opaque surface, describing the relationship between the incoming light direction and the outgoing reflection direction. The BRDF is essential in realistic rendering, as it helps determine the colour and intensity of reflected light, contributing to the material's appearance under different lighting conditions. 20, 21

buffer: In computer graphics, a buffer is a temporary holding area in memory, typically on the GPU, used to store data (like voxel information) or intermediate calculations (like depth values) during rendering. This ensures smooth data flow and optimizes the process. 9, 18, 27, 28, 29

BVH **– Bounding Volume Hierarchy**: A tree structure used in computer graphics and computational geometry to accelerate spatial queries, such as ray tracing and collision detection. Each node in the hierarchy represents a bounding volume that encapsulates a set of objects or smaller bounding volumes, allowing for efficient testing and exclusion of large groups of objects in complex scenes. 18, 19, 34

cellular automata: Mathematical models consisting of a grid of cells, each in one of a finite number of states, that evolve through discrete time steps according to a set of rules based on the states of neighbouring cells. Used to simulate complex systems and processes in fields like computer science, physics, and biology. 33

colour correction: The process of adjusting the colour balance, contrast, and saturation of a rendered image to achieve a desired visual appearance. This can involve fixing colour casts, enhancing realism, or creating a specific artistic style. 30

CPU **– Central Processing Unit**: Processor of the computer where most of the computation happens. 10, 15, 18

CUDA: A platform that allows to write C++ programs that run on NVIDIA GPUs. 25

deferred rendering: A graphics technique where scene geometry is first rendered into intermediate buffers , then lighting calculations are performed separately, allowing for efficient handling of complex lighting effects in real-time graphics applications. 27, 28

depth of field: The portion of an image that appears sharp, ranging from the near focus point to the far focus point. Controlled by aperture, focal length, and distance to the subject. 30

forward rendering: A real-time rendering technique where objects are individually rendered with lighting calculations applied during the process. It is suitable for scenes with fewer dynamic lights but can become computationally intensive with more lights or objects. 28

gamma correction: A technique that adjusts the brightness of colours in an image to compensate for the non-linear response of monitors. This ensures images appear visually consistent with how we perceive light, making dark areas appear darker and bright areas appear brighter. 30, 32

GPU – **Graphics Processing Unit**: Also commonly called graphics card (although they are different). It is responsible for computing graphics related tasks. 10, 15, 17, 18, 23, 30, 32, 34, 35, 48, 57

ID – **identifier**: Unique label or value assigned to distinguish individual entities within a system or dataset. 18

integrated GPU: On laptops that require a powerful graphics card, such as gaming laptop, they often have a dedicated Graphics Processing Unit (GPU) (often NVIDIA or AMD) and an integrated GPU , also called discrete GPU (often Intel or AMD) that is located on the Central Processing Unit (CPU) . This is mainly for power efficiency. 24

LLVM: A collection of modular and reusable compiler and toolchain technologies for developing compilers and related tools. It optimizes code at various stages, supports multiple languages and platforms, and generates high-performance machine code. 25

LOD – **Level Of Detail**: Technique used to improve performance in video games. The idea is that distant objects are replaced by lower details version since they will be less visible. 17, 18, 24, 33

mesh: A digital representation of a three-dimensional object composed of vertices that make up triangles. 12, 15, 22

mipmap: A series of pre-generated textures derived from an original texture, each at a progressively lower resolution. They enhance rendering efficiency and image quality by providing optimized textures for objects viewed at different distances or sizes. 28

motion blur: A visual effect simulating the blurring of objects in motion, mimicking real-world camera exposure time and enhancing the perception of speed. 30

OS – **Operating System**: Integral software that governs computer hardware, enabling application execution, resource allocation, and user interaction through services like memory management, process scheduling, and file handling. 15

PBR – **Physically Based Rendering**: A rendering technique that simulates the physical properties of light and materials for realistic lighting and shading effects, making objects appear more natural. 20

PDF – **Probability Density Function**: A function that describes the likelihood of a continuous random variable taking on a specific value, where the area under the curve represents the probability of the variable falling within a given range. 21

shader: A program that runs on the GPU to manipulate the rendering of graphics primitives, controlling aspects like vertex transformations, lighting, and pixel colour generation. 15, 18, 19, 23, 24, 28

stack: Data structure where you can only add or remove items from the top. It follows the Last In, First Out (LIFO) principle, like a stack of plates. 17

stochastic: Refers to systems or processes characterized by inherent randomness, introducing uncertainty into calculations or simulations across fields like mathematics, statistics, and computer science, often employing probabilistic models to account for variable factors. 29

streaming: The process of continuously transmitting data over a network or from a storage device to a receiving device, allowing playback or use of the data without the need for downloading the entire file beforehand. 17, 18, 25

***TAA* – Temporal Anti-Aliasing**: An anti-aliasing technique that improves image quality by combining information from current and previous frames to reduce aliasing (jagged edges). This method offers performance benefits compared to traditional anti-aliasing methods. 30

texel: The smallest unit of a texture map, representing a single point of texture data applied to a 3D model's surface in computer graphics. If a pixel is a texture element, a texel is a texture element. 29, 33

tone mapping: Tone mapping adjusts the wide range of lighting values from the rendered scene to a narrower range suitable for display on monitors with a limited dynamic range. This preserves details in highlights and shadows while creating a natural-looking image. 30, 32

1 Introduction

Minecraft (2011), the most bought game with three hundred million copies sold (Boddy, 2023), has the player explore and build in a voxel world. In 2020, *Teardown* was released featuring ray tracing and a fully destructible voxel world, selling almost three million units (VG Insights, no date).

Voxels are often imagined and portrayed as cubes, but as it will be explained later, it is more accurate to understand them as being a value on a regular grid.

Lighting computation has long been a challenge for interactive graphics. Direct lighting can be computed in real-time, but indirect lighting is usually harder to calculate and therefore pre-computed. This is a technique called Lightmaps introduced in *Quake* (1996) (Abrash, 2000). Combining direct lighting and indirect lighting leads to having global illumination. It is often accomplished using ray tracing. The discrete nature of voxels allows the simplification of the problem of ray intersection tests. Voxels are even used with traditional meshes for global illumination by voxelizing the mesh data (Crassin *et al.*, 2011).

There are other uses for voxels. For example, in medical imagery, MRI and ultrasound scans are stored using them. *The Legend of Zelda: Tears of the Kingdom* (2023) has a voxel representation of the world. They are not directly displayed, but used for gameplay elements and to simulate sound propagation (Dohta, Osada and Takayama, 2024).

The primary goal of this bachelor will be to implement a voxel ray tracing in a Vulkan application using the Jai programming language. This will be done in a compute shader, which in part justifies the choice of Vulkan as a graphics Application Programming Interface (API) . Its predecessor, OpenGL, does not natively support compute shaders and would have required to either use OpenCL, or do the work in a fragment shader. DirectX could have been used, but it is not cross-platform and is not an open standard.

The plan of this thesis is the following: first a state of the art about voxel rendering will be presented. It will contain various definitions, voxel storage techniques and examples in games. Then, in the qualitative analysis part, a game and an engine will be analysed more in depth and the result of interviews done with experts will be presented. These are *Teardown* and *GVox Engine*. The former because it features a fully destructible voxel world with ray tracing and is a successful commercial game. The latter because it is an open-source voxel engine with path tracing and has world stored using brickmaps, which is a data structure presented in Section 2.2.2.4. After that, the test protocol of the media project will be explained. The key metrics are frame time, memory consumption and the amount of voxels. This will be followed by a production overview of the media project. The quantitative analysis will then be used to examine the measurements done the media project.

2 State of the Art

This section will present various techniques used to render voxels. Then, it will show how they are applied to various projects.

2.1 Definitions

The term voxel comes from the contraction of the words “volume” and “pixel”. Pixel itself comes from “picture element”. We can then understand voxel as being a “volume element” (Akenine-Möller *et al.*, 2018). Although they are often portrayed as cubes (see Figure 1), it is more accurate to think of them as a value on a regular grid in 3D space.

Figure 1: Screenshot of the game *Minecraft*, which uses voxels. (Mojang, 2011)

Using voxels in games is not new: in 1998, after the development of *Quake II* (1997), John Carmack explained that he tried using them for *Quake III* (1999). He decided against it since doing so was too slow without the use of specialized hardware (Abi-Chahla, 2009).

Indeed, graphics cards are very good at displaying triangles (Akenine-Möller *et al.*, 2018), and at the time you couldn’t run arbitrary code for each pixels like today. However, this restriction lead developers to directly use rasterisation to render voxels. Rasterisation, in this context, is the process by which a triangle (or any other primitive shape) is transformed to a series of pixels. This method is explained in more depth in Section 2.2.1.

But this is not what Carmack tried for *Quake III*. He was using a technique called ray casting. The idea is that instead of transforming 3D triangles to 2D and then rasterising them, rays are shoot from the camera to check if they collide with some geometry. The geometry can be of any kind; for example, *Wolfenstein 3D* (1992), also worked on by John Carmack, casts rays against a 2D grid and uses the distance of those rays to simulate 3D (Vandevenne, 2004). See Figure 2. In the context of voxel rendering, it would mean casting rays against a voxel grid.

Figure 2: Screenshot of the game *Wolfenstein 3D*, which uses ray casting to simulate 3D. (id Software, 1992)

Recently, there has been an increase in interest in another technique by the name of ray tracing. Sometimes it is used interchangeably with ray casting, but it is more helpful to consider ray casting as some way of doing ray tracing. Ray tracing is generally used to refer to the act of using rays to simulate the behaviour of light. There are multiple methods for that, one of which is path tracing.

As previously stated, path tracing is a way of doing ray tracing. Its specificity is that it will recursively bounce rays in a way that will give a realistic feeling to the lighting of a scene. This algorithm allows for Global Illumination (GI), which is when the lighting is not only considered directly (direct illumination) but indirect bounces are also used (indirect illumination). This can be observed in Figure 3. The difference between ray casting, ray tracing, and path tracing is illustrated in Figure 4.

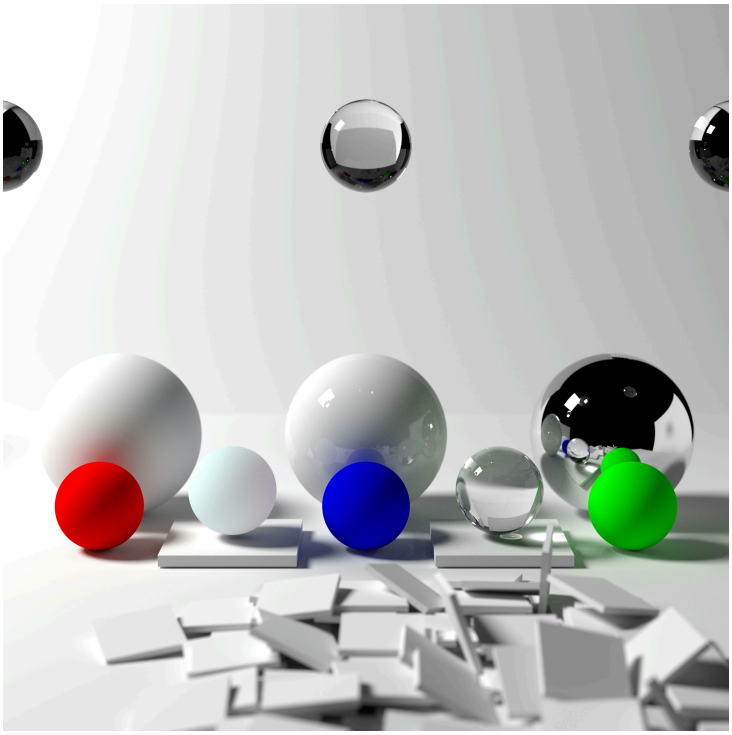


Figure 3: An image rendered using path tracing, demonstrating notable features of the technique. (Wikipedia Contributors, 2024a)

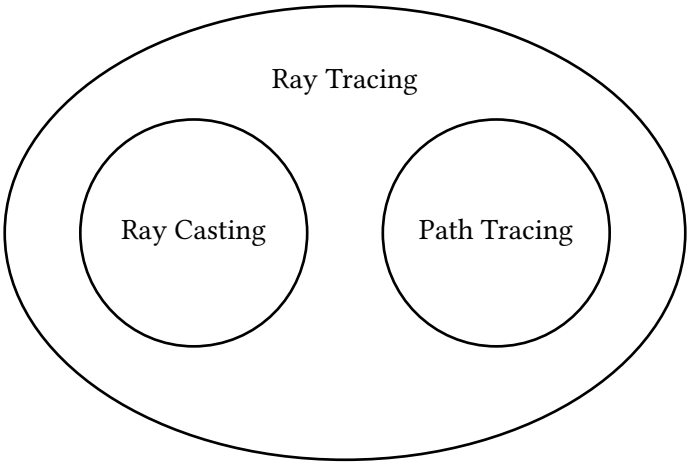


Figure 4: Illustration of the classification of ray casting, ray tracing and path tracing.

To be able to use GPUs , developers can use graphic APIs . These abstract the underlying machine, to allow to write software once for multiple graphics cards. Some of these APIs come from the Operating System (OS) , such as DirectX on *Windows* and Metal on *MacOS*. On the other hand, open standards have been developed to have cross-platform graphic APIs . OpenGL was created in 1992 by Silicon Graphics (Astle and Hawkins, 2004) but was then transferred to the Khronos Group in 2006 (Wikipedia Contributors, 2024b). Its successor Vulkan, which was originally called OpenGL Next, was released in 2016. It is meant to be a lower-level API and allows for more control over the GPU . Most importantly for voxel rendering, it allows the use of compute shaders directly, which was not the case with OpenGL since it required to use OpenCL for that (Wikipedia Contributors, 2024c). Apart from surface extraction explained in Section 2.2.1, voxel rendering techniques usually require the use of compute shaders since they are not based on the traditional triangle pipeline.

2.2 Current Voxel Rendering Techniques

Multiple methods to render voxels have been developed throughout the years, both on the CPU and on the GPU . This was pushed by both medical imagery and video games.

2.2.1 Surface Extraction

This method is the most commonly used one in video games; it is notably used by *Minecraft*. The idea is that the surface of the voxel data is first converted to a triangle mesh before being sent to the GPU . Just turning every voxel into a cube would be inefficient as most of the triangles would be hidden by neighbouring voxels. Therefore methods are usually used to reduce the number of triangles.

The first one is to check around the current voxel if it has any neighbours. If yes, the face between the current voxel and the neighbour will not be meshed. Additionally, other methods can be used, such as greedy meshing, which will look for neighbouring voxels of the same type and will extend the triangles to them (Lysenko, 2012). This is illustrated in Figure 5.

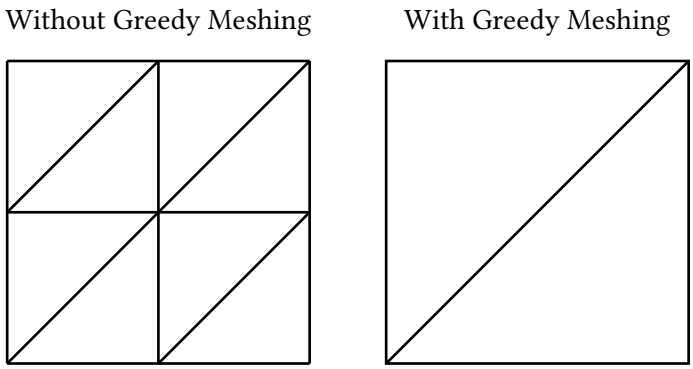


Figure 5: 2D example of greedy meshing. In 3D, it is done on each axis separately, so the principle is the same.

There are various other ways to accelerate the rendering of voxels using rasterisation, such as culling, vertex compression and instancing but these will not be explained here as it is outside the scope of this study.

Using triangles may have the advantage of being able to use Hardware Ray Tracing (HWRT) since it is designed with the traditional graphical pipeline in mind. But using custom intersection code can allow to do HWRT without triangles.

2.2.2 Volume Marching

Volume marching is the family of techniques that use rays that traverse some kind of data structure. So they all use ray tracing and each data structure has its pros and cons.

2.2.2.1 Fast Voxel Traversal Algorithm

In their paper, Amanatides and Woo (1987) present an algorithm for traversing voxels using rays without checking the collision against every one of them. This allows to only check voxels that are traversed by the ray. However, this method stores the voxels in a 3D grid, meaning that if there is a significant amount of empty space, the ray still has to check collisions against each empty voxel. This is illustrated in Figure 6.

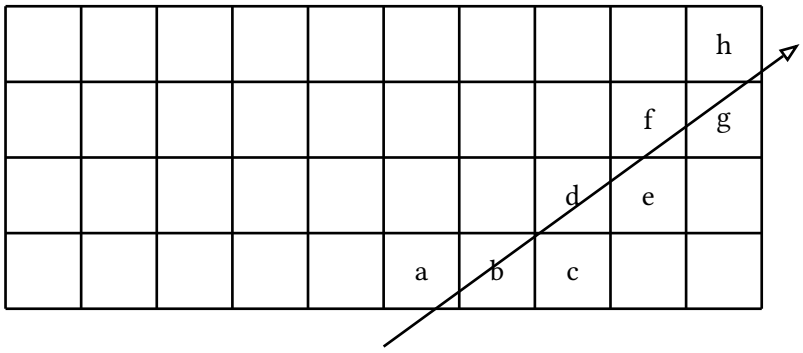


Figure 6: Illustration of the Fast Voxel Traversal Algorithm. Instead of checking the whole grid for a collision, it allows us to only check voxels a, b, c, d, e, f, g and h.

This algorithm is at the heart of every other voxel ray tracing method, so although it is rarely used by itself, it was improved later on. This way of traversing voxels is also commonly called a Digital Differential Analyser (DDA).

2.2.2.2 Sparse Voxel Octrees

Sparse Voxel Octrees (SVOs) are a way to profit from the sparsity of most voxel data sets. The idea is to group voxels in a big voxel that is itself part of a bigger voxel. The advantage comes from the fact that if a parent node does not have children, it can just be marked as empty and the traversal does not have to check inside it. This is illustrated in Figure 7.

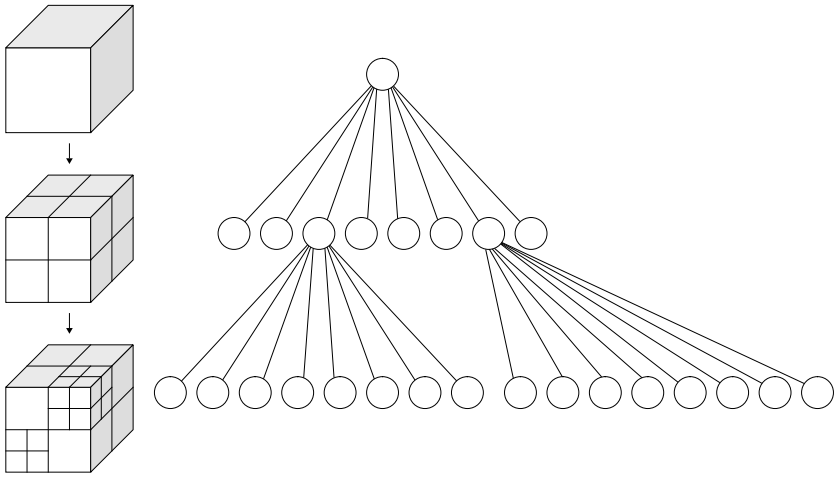


Figure 7: Schematic drawing of an octree. (Wikipedia Contributors, 2024d)

This technique was explored by Laine, Marton and Gobbetti (2010). In their paper, they present a ray cast algorithm that traverses the SVO using a stack . As they explain, this is quite costly for a GPU , as their architecture is not well suited for that kind of data structure. Another drawback of SVOs is that they are quite costly to update, which is not ideal for video games since interacting with the world is a big part of what makes voxel games attractive.

2.2.2.3 Sparse Voxel Directed Acyclic Graphs

Sparse Voxel Directed Acyclic Graphs (SVDAGs) can be seen as a modification of SVOs. They try to reduce the memory consumption of SVOs by finding similar nodes in the tree and allowing a child node to be referenced by multiple parents. This has the effect of reducing repetitions in that data structure. You can see a comparison of SVOs and SVDAGs in Figure 8.

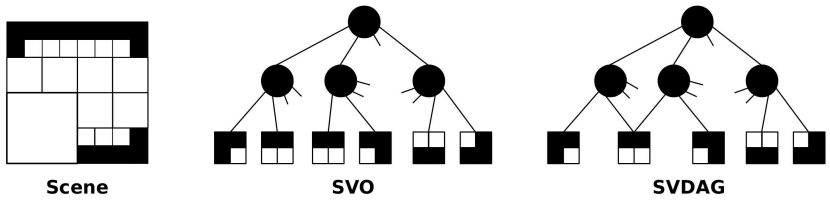


Figure 8: Comparison of SVOs and SVDAGs. (Villanueva, Karras and Gobbetti, 2017)

This comes with the drawback that if you modify a voxel that is in a node referenced by two parents, it can be quite computationally intensive to update the SVDAG. This is a problem that Careil, Billeter and Eisemann (2020) have tried to address in their paper. They made an interactive prototype application where you can edit scenes at a resolution of $128'000^3$.

2.2.2.4 Brickmaps

As previously mentioned, SVOs and SVDAGs have the disadvantage of slowing down the edition of the voxel grid with the trade off of accelerating the rendering. This is something that van Wingerden, T. (2015) has tried to solve using a novel approach called Brickmaps. This technique also provides a solution to streaming the voxel data to the GPU when it is needed as well as Level Of Detail (LOD) system.

This data structure works by having multiple levels of grids. The cells of each level point to multiple cells of the lower level until the bottom level where the voxels are stored using bit masks and some colour information. Since a higher-level cell can either point to more data or be empty, brickmaps are similar to SVOs as they allow to skip empty space efficiently.

The bottom level grid is called the brickmap. It has a size of 8^3 and is the one that stores the voxel data. One level higher is the brickgrid, where the cells either point to a brickmap or are empty. It can be of any size. Finally, there is the higher-level grid, which points to a group of brickgrid cells. It is possible to have multiple levels of higher-level grids. This structure is illustrated in Figure 9. The brickmap is in blue and the brickgrid in red. The filled squares in the grid are non-empty cells. The green grid is the higher level grid. Each grid level is traversed using the DDA algorithm by Amanatides and Woo (1987).

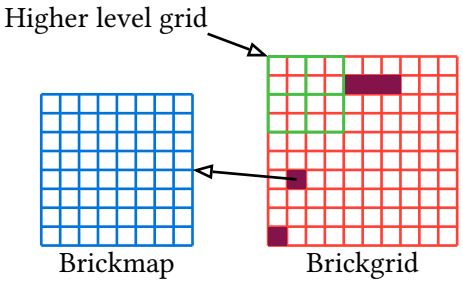


Figure 9: Illustration of how brickmaps work.

As outlined earlier, brickmaps have a way to efficiently handle LODs and data streaming . To do that, each brickgrid and higher-level cells store an LOD colour. This value is used for rendering when the lower-level is either not loaded or too small to profit from the finer details. When a voxel is rendered using an LOD colour because the lower-level is not loaded, the cell identifier (ID) is put in a shared buffer between the CPU and the GPU . This is then used after the rendering of the scene to know which data should be loaded.

2.2.2.5 Hardware Ray Tracing (HWRT)

In 2018, *NVIDIA* unveiled their Turing architecture (Kilgariff *et al.*, 2018), being the first series of graphics cards for the general public with hardware-accelerated ray tracing. They came with a new set of tools in graphics APIs that allowed programmers to use these cores, namely DXR (DirectX Raytracing) in DirectX 12 (D3D Team, 2018) and the Vulkan Ray Tracing Extensions in Vulkan (Koch *et al.*, 2020). They allow access to five new shaders (Akenine-Möller *et al.*, 2018):

1. Ray generation shader
2. Closest hit shader
3. Miss shader
4. Any hit shader
5. Intersection shader

The hardware ray tracing pipeline has features to accelerate the collision check of rays against the data. This is done using acceleration structures separated into two parts, the Top-Level Acceleration Structure (TLAS) and the Bottom-Level Acceleration Structures (BLAS). There is only one TLAS, but there are multiple BLAS. The user has no control over the TLAS, as it is implementation-defined. Yet, it is often implemented using a Bounding Volume Hierarchy (BVH) . There are two kinds of BLAS: triangle groups and analytical/procedural. Procedural BLAS are defined using an intersection shader . This is illustrated in Figure 10.

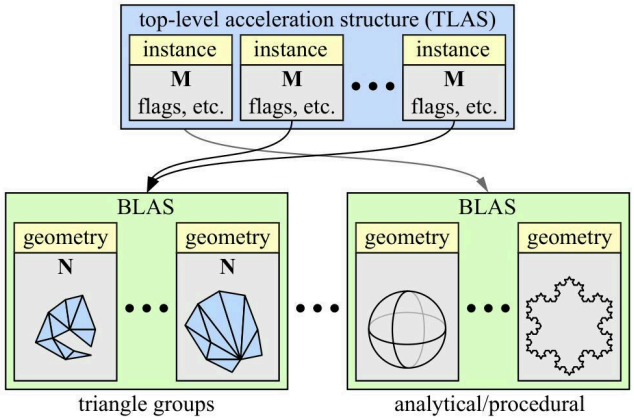


Figure 10: Illustration of how the Top-Level Acceleration Structure (TLAS) is connected to a set of Bottom-Level Acceleration Structures (BLAS). (Akenine-Möller *et al.*, 2018)

There have been few projects using hardware ray tracing to render voxels. Recently, Elwell (2024), a notable voxel project author and YouTuber, exposed his method in a video. In it, he explains that he created a custom intersection shader that is tasked with checking the collision of a ray against a voxel volume using the DDA algorithm. The BLAS contains multiple voxels. This means that it is possible to move these volumes continuously if that is desired.

A potential drawback of using HWRT for voxel rendering is that you're profiting less from the speed that ray tracing voxels has compared to data like triangles. That is because the ray has to first traverse the TLAS which is likely to be a BVH before being able to check a collision against a voxel. On the other hand, the BVH could have the advantage of helping sparse voxel data, as empty space could just be skipped. Whether the ray collision check is accelerated by the BVH or not, there is always the cost of building it to take into consideration.

2.3 Global Illumination with Path Tracing

Computing global illumination using recursive techniques was pioneered by Whitted in 1979 in his paper *An improved illumination model for shaded display*. His technique allows to have true shadows, reflections and refraction as illustrated in Figure 11.

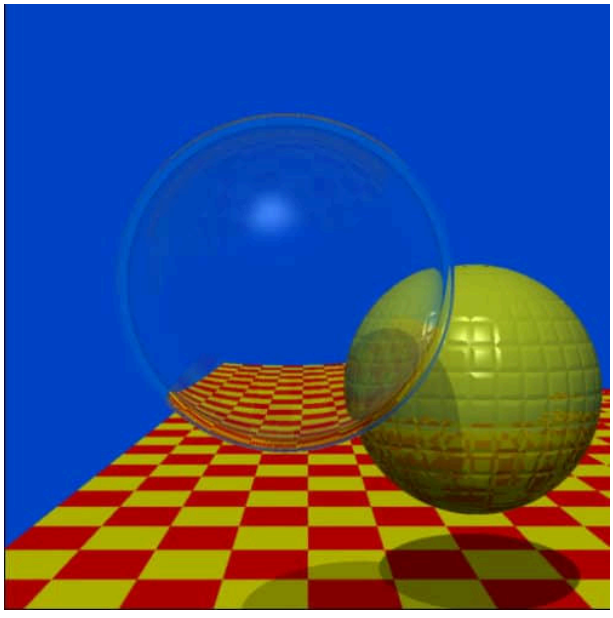


Figure 11: Image rendered using Whitted style ray tracing (Whitted, 1979)

His model may be written as

$$I = I_a + k_d \sum_{j=1}^{j=\text{ls}} \left(\vec{n} \cdot \vec{l_j}\right) + k_s S + k_t T \tag{1}$$

where

- I is the light intensity
- I_a is the ambient light
- k_d is the diffuse reflection constant
- $\sum_{j=1}^{j=\text{ls}}$ is the sum of every light source in the scene
- $\vec{n}$ is the normal of the surface
- $\vec{l_j}$ is the light direction
- k_s is the specular reflection coefficient
- S is the intensity of light incident from the $\vec{r}$ direction (reflection direction)
- k_t is the transmission coefficient
- T is the intensity of light from the $\vec{p}$ direction (refraction direction)

k_s and k_t where constants for the image in Figure 11 but they should be a function that incorporates the Fresnel reflection law. As for the direction $\vec{p}$, it should obey Snell's law.

Although this model allows to have some optics effects, it is not physically based. Cook and Torrance (1982) have proposed a model based on microfacets theory in their paper *A Reflectance Model for Computer Graphics*. According to de Vries, for a lighting model to be considered physically based it needs to: "(1) Be based on the microfacet surface model (2) Be energy conserving (3) Use a physically based Bidirectional Reflectance Distribution Function (BRDF)"

The goal of modern Physically Based Rendering (PBR) is to compute the rendering equation presented by Kajiya, J. T. (1986). This equation may be written as

$$\begin{aligned} L_o(\vec{p}, \vec{\omega}_o) &= L_e(\vec{p}, \vec{\omega}_o) + L_r(\vec{p}, \vec{\omega}_o) \\ L_r(\vec{p}, \vec{\omega}_o) &= \int_\Omega f_r(\vec{p}, \vec{\omega}_i, \vec{\omega}_o) L_i(\vec{p}, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) \, \mathrm{d}\vec{\omega}_i \end{aligned} \tag{2}$$

where

- $L_o(\vec{p}, \vec{\omega}_o)$ is the total radiance (which will be explained later) at a point $\vec{p}$ at an outgoing direction $\vec{\omega}_o$
- $\vec{p}$ is the point is space
- $\vec{\omega}_o$ is the direction of outgoing light
- $L_e(\vec{p}, \vec{\omega}_o)$ is the emitted radiance
- $L_r(\vec{p}, \vec{\omega}_o)$ is the reflected radiance
- $\int_\Omega \dots \mathrm{d}\vec{\omega}_i$ is an integral over the hemisphere Ω
- Ω is the hemisphere around the normal $\vec{n}$ that contains every values for $\vec{\omega}_i$ where $\vec{\omega}_i \cdot \vec{n} > 0$
- $f_r(\vec{p}, \vec{\omega}_i, \vec{\omega}_o)$ is the BRDF or the amount of light that is reflected from $\vec{\omega}_i$ to $\vec{\omega}_o$ at point $\vec{p}$
- $\vec{\omega}_i$ is the opposite direction of the incoming light
- $L_i(\vec{p}, \vec{\omega}_i)$ is the radiance arriving at $\vec{p}$ from the direction $\vec{\omega}_i$
- $\vec{n}$ is the normal of the surface at point $\vec{p}$
- $(\vec{\omega}_i \cdot \vec{n})$ is the attenuation factor of the irradiance due to the incident angle, which takes into account the spreading of the radiance on the surface when the lighting is not perpendicular. It can be simplified to $\cos\theta_i$ and is often called the geometry term.

Radiance can be defined as the radiant flux emitted, reflected, transmitted or received by a given surface, per unit solid angle per unit area. The solid angle, noted as ω, is the area of a shape projected onto a unit sphere. The area is the one in which point $\vec{p}$ lies. Radiant flux, noted as Φ, is the total energy of a light source over every wavelength.

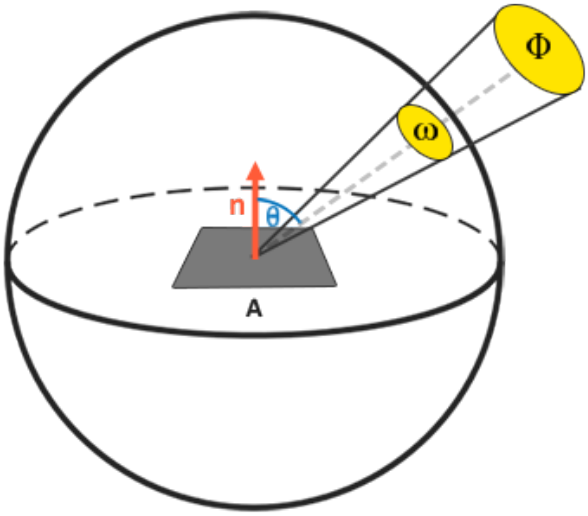


Figure 12: Illustration of radiance over an area A and the solid angle ω. (de Vries, J., 2014)

Since we want all incoming light, we need to have the value of radiance over every solid angle value. This is called irradiance and is therefore defined as the radiant flux received by a surface per unit area. However, if we think of how the integral can be computed, we can, for example, sum up the incoming light over the hemisphere. In this context, we would only consider one direction on the hemisphere at a time, which would be the radiance and then, once this energy is projected onto the surface, it becomes irradiance. In the rendering equation, this is done by multiplying by the geometry term. To conclude, $L_i(\vec{p}, \vec{\omega}_i)$ is the incoming radiance, $L_i(\vec{p}, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n})$ is the irradiance and then the BRDF is applied to the irradiance.

One way of solving this equation using ray tracing would be to shoot rays from our surface towards light sources, a point light for example, and check if the surface should be lit, determined by if a ray hits something before hitting the light. Point lights represent an infinitely small light source, which does not exist in real life, and gives hard shadows. One way to remedy that, would be to have a light that has an area. Points would then be randomly generated on this area and rays can be traced towards them. This would give smooth shadows, but requires to shoot more than one ray, and the more rays are shot, the less noisy the shadows will be. This process is called ray-traced shadows.

This process only considers direct lighting and is therefore not path tracing. To have indirect lighting, we must compute the integral in a recursive manner, until reaching the sky or the sun or if the maximum number of bounces is reached. With this technique, rays are shot from the camera and upon hitting a surface, a random direction is chosen on the hemisphere directed along the surface normal (Wikipedia Contributors, 2024a).

If you have an equal chance of picking any value over the hemisphere, the sampled value must be divided by a Probability Density Function (PDF) which, in this case, is a constant equal to $\frac{1}{2\pi}$ since 2π is the area of the hemisphere (Bikker, 2024a).

One way to reduce noise is to use importance sampling using a cosine weighted diffuse direction. The PDF must then be changed to $\frac{\vec{n} \cdot \vec{r}}{\pi}$ where $\vec{r}$ is the cosine weighted random diffuse direction. Pseudocode for a simple path tracer without reflection and refraction can be seen in Listing 1.

```glsl
vec3 Trace(Ray ray) {
  const vec3 sky_colour = vec3(1.2, 1.2, 1.0);
  vec3 throughput = vec3(1.0);
  for (int bounce = 0; bounce < 4; bounce++) {
    FindResult find_result = FindNearestObject(ray);
    if (find_result.is_empty) return throughput * sky_colour;

    vec3 brdf = find_result.base_colour / PI;
    vec3 new_direction = CosWeightedDiffuseReflection(find_result.normal);
    float pdf = dot(find_result.normal, new_direction) / PI;
    throughput *= brdf * (dot(find_result.normal, new_direction) / pdf);
    ray = Ray(find_result.collision_point, new_direction);
  }

  return vec3(0.0);
}
```

Listing 1: Pseudocode for a simple path tracer.

2.4 Voxel in Video Games and Existing Engines

This section contains a brief explanation of how some projects have chosen to render voxels. *Teardown* and *GVOX Engine* are analysed more thoroughly respectively in Section 3.1.1 and Section 3.1.2.

RenderDoc, an open-source graphics debugger for capturing and analysing frames in graphics applications, was used to see how some of these games work.

Minecraft sparked significant interest in voxels. As can be seen in Figure 13, it uses surface extraction as a rendering technique. To handle its infinite maps, the world is split into chunks, that are each rendered separately. In the figure, you can see the draw call of a chunk with its mesh data.

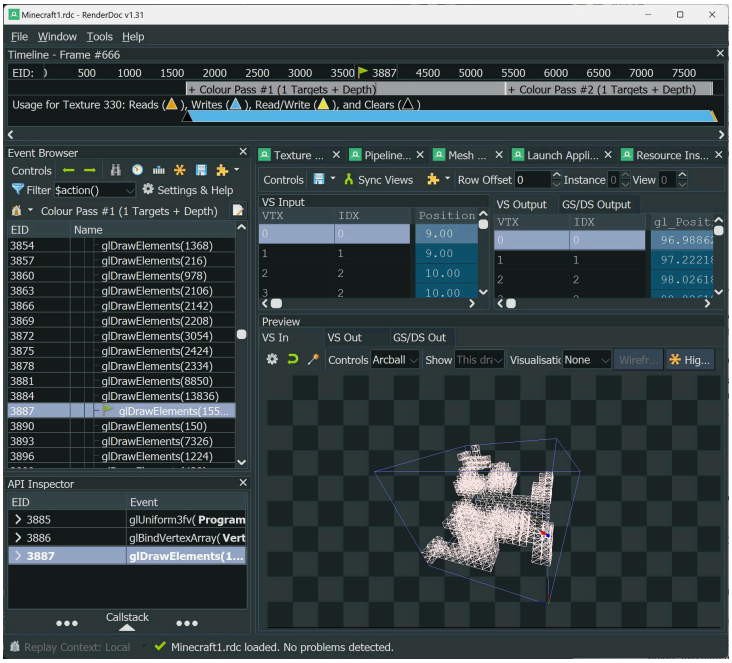


Figure 13: Screenshot of a *RenderDoc* capture of *Minecraft*.

To handle textured voxels, *Minecraft* uses a texture atlas, as this allows to render an entire chunk at once without having to render each voxel type individually. This has the consequence of not being able to use greedy meshing as you can't use texture repetition.

To handle lighting, each empty voxel has a lighting value between 0 and 16 that decreases when not exposed to the sky or during the night.

Since *Minecraft*'s lighting model is pretty simple, users have created mods that allow the creation of custom shaders . With them, programmers have been able to create more complex lighting as can be seen in Figure 14. These shaders are implemented using the traditional graphical pipeline. This is presented in a video by Acerola (2021).

Figure 14: Screenshot of *Minecraft* with the *Sildur's Vibrant Shaders*.

Cube World (von Funck, W. and von Funck, S., 2019) is an RPG well known for its voxel look. The techniques the game utilizes are similar to the ones used in *Minecraft*. It uses surface extraction and sends the data to the GPU in chunks (see Figure 15). It also does not use greedy meshing. Where it differs from *Minecraft* is that instead of using textures, it uses vertex colours, so the voxels all have a uniform colour.

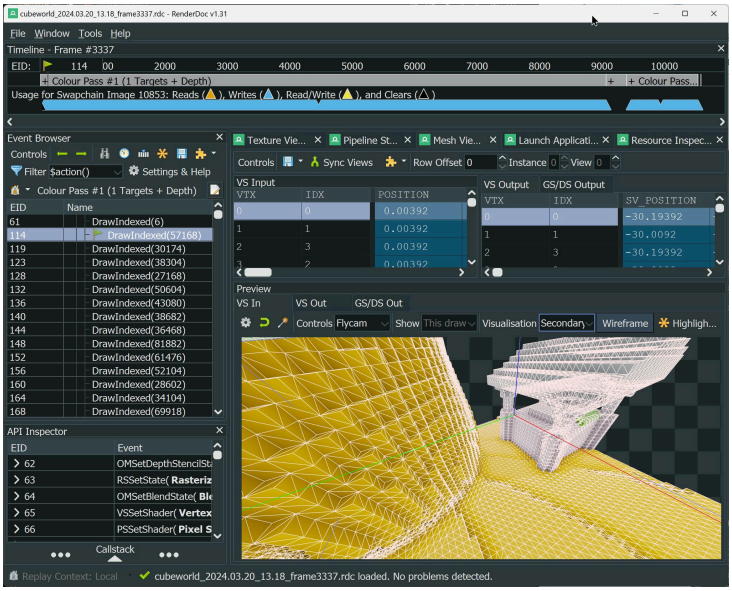


Figure 15: Screenshot of a *RenderDoc* capture of *Cube World*.

Outside of finished games, some people have started to write voxel engines and share their progress on the Internet. One example is *Ethan Gore's Voxel Project*, which has a very high level of detail. Ethan Gore hasn't disclosed many details on how this is achieved, and the engine is closed source. However, he mentioned in an FAQ that he does use rasterisation and not ray tracing (Gore, 2024), emphasizing that significant gains derive from a robust LOD system. Using *RenderDoc*, it is possible to gain more information. It seems like the drawing of voxels is separated into chunks which are also separated by faces. This means that the engine starts by drawing the faces that are in the direction of the up axis and then continues with the other ones. This is visible in Figure 16. You can see that some chunks are not yet drawn and that the engine does not draw every axis at the same time.

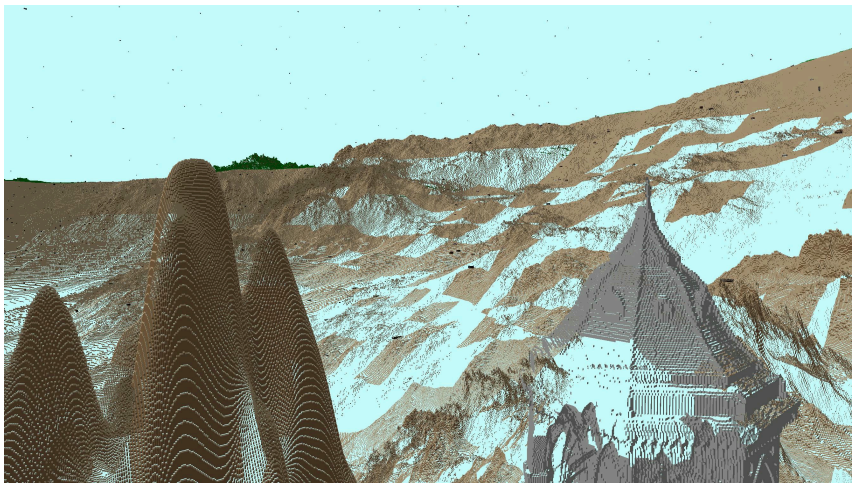


Figure 16: Image taken in the middle of the rendering of a frame in *Ethan Gore's Voxel Project*.

Another voxel engine that is often showcased in *YouTube* videos is *Octo Voxel*. In one of these, Dwyer (2022) explains the rendering techniques he uses. His goal was to be able to run the engine on integrated GPUs , so although he started with a ray tracing-based renderer, he switched to a technique he called "parallax ray marching". It works by rendering boxes through the rasterisation pipeline and then doing ray tracing in the fragment shader . This is efficient because you only need to check for collisions inside that box, and thus allows to skip empty space. However, it has the limitation of not allowing to bounce rays around the scene. So lighting effects need to use rasterisation techniques such as shadow mapping.

Octo Voxel was recently switched back to ray tracing to avoid these limitations. Although the switch is not complete, the new renderer was showcased in a video where Dwyer (2024) shows it rendering scenes using ray tracing to test for shadows. This can be seen in Figure 17 under the trees.

Figure 17: Screenshot of the latest version of *Octo Voxel* using ray tracing as a rendering technique. (Dwyer, 2024)

2.5 Media Project Objectives

The media project of this bachelor will be about implementing a voxel renderer using Vulkan and Jai. The goal is to have a desktop application that can run on both Windows and Linux using the cross-platform abilities of Vulkan. It would be able to load voxel scenes that are in the `.gvox` format (Rundlett and Dwyer, 2022). If this turns out to be too hard, it is possible that the project uses the `.vox` format (ephtracy, 2015). And if that also fails, it is possible that a hard coded version will be used. Ideally, it would demonstrate the ability to use advanced rendering techniques that are possible using ray tracing such as global illumination (or some form of indirect illumination).

The Jai programming language is a project started by Jonathan Blow in 2014 as a C++ replacement to support his video game project, Sokoban. The Jai compiler has significantly expanded since its inception, incorporating features like an LLVM backend, macros, and inline assembly support. As of may 2024, the closed beta includes over 600 members. Jai aims for high performance, targeting the compilation of 1 million lines of code in under one second, currently achieving 250,000 lines per second with the x64 backend. Jai is an imperative, statically typed language offering metaprogramming, multiple return values, and operator overloading, while excluding elements like garbage collection, exceptions, and object-oriented inheritance. Supporting Windows, Ubuntu Linux, MacOS and others, Jai's syntax is still provisional, with refinements anticipated before its official release alongside Blow's game, Sokoban.

The technique that will be used to store and render voxels will start by a simple array. Ideally, it would be upgraded to brickmaps, explained in Section 2.2.2.4. If this succeeds, it will be extended to allow the usage of different kinds of materials, which is necessary for global illumination, similar to the system introduced by Hjerpbakk (2022).

It is uncertain if it will be possible to have global illumination in the time available for this project. Furthermore, as the Brickmap paper was programmed in CUDA , the Vulkan implementation may prove to be more difficult, especially on the data streaming part.

The media project is planned to use these libraries:

- **SDL2**: Cross-platform window management
- **Vulkan Memory Allocator**: Easy to integrate Vulkan memory allocation library
- **ImGui**: Allows to easily create GUI tools
- **Gvox**: Allows to load `.gvox` files that contain voxel models

Some of these have wrappers provided with the Jai compiler, but others may require to write bindings for. However, Jai comes with a tool to automatically generate bindings from C/C++ headers.

To test the media project, multiple variables will be measured. Performance being the most important one, frame time (ms) will be timed. A challenge with voxel data is memory consumption; this will also be measured. The compression ratio of a data structure impacts its building performance, so along with traversal speed, building speed will be measured brickmaps are implemented. It will be compared to other voxel engines if possible and pertinent. It is also possible that the media project will be compared to itself before and after a specific optimization has been applied.

2.6 Summary

In this section, we looked at definitions that are important to know in the context of voxel ray tracing, such as ray casting and path tracing. We also looked at the two main families of voxel rendering in the name of surface extraction and volume marching. We then looked at papers that explain how to implement ray tracing, such as the fundamental Fast Voxel Traversal Algorithm. An overview of the use of HWRT was also studied. Then, the rendering techniques of various games and engines were

explained and analysed, followed by a presentation of the planned media project which was helped by the former analysis.

3 Qualitative Analysis

This section will contain an analysis of two notable voxel projects, *GVOX Engine* and *Teardown*, and then a presentation of interviews done with experts in voxel rendering. This will help us to fulfil the objectives of the media project.

3.1 Voxel Renderers Analysis

3.1.1 Teardown

Teardown is a sandbox puzzle game released by Tuxedo Labs (2022). It uses a custom voxel engine made by Dennis Gustafsson. This is an interesting project to analyse because it contains levels made of destructible voxels. It also features fully dynamic lighting that uses ray tracing. Furthermore, the objects don't have to align to the voxel grid, because it is also based on a technique similar to parallax ray marching in *Octo Voxel* as described in Section 2.4.

Figure 18: Screenshot of the game *Teardown*. (Tuxedo Labs, 2022; Wittens, 2023)

This analysis is based on three sources: a *Twitch* stream presented by Dennis Gustafsson (Tuxedo Labs, 2020) and two blog posts on the rendering of *Teardown*. Namely *Teardown Frame Teardown* (Wittens, 2023) and *Teardown Teardown* (Ryan, 2022).

Teardown uses deferred rendering , meaning that it first starts by rendering the geometry buffer (G-buffer). Its layout is described in Table 1 and a breakdown on how to make the image in Figure 18 is available in Figure 19.

	R	G	B	A	
Base colour	Red	Green	Blue		`RGB16_UNORM`
Normal	Normal X	Normal Y	Normal Z		`RGB8_SNORM`
Material	Reflectivity $[0, 1]$	Smoothness $[0, 1]$	Metallic $[0, 1]$	Emissive $[0, 32]$	`RGBA16_FLOAT`
Motion	Motion X	Motion Y	Water Mask		`RGB16_FLOAT`
Depth	Linear Z				`R16_UNORM`
Z-buffer	Hyperbolic Z				`D32_UNORM`

Table 1: Structure of the G-buffer of *Teardown*.

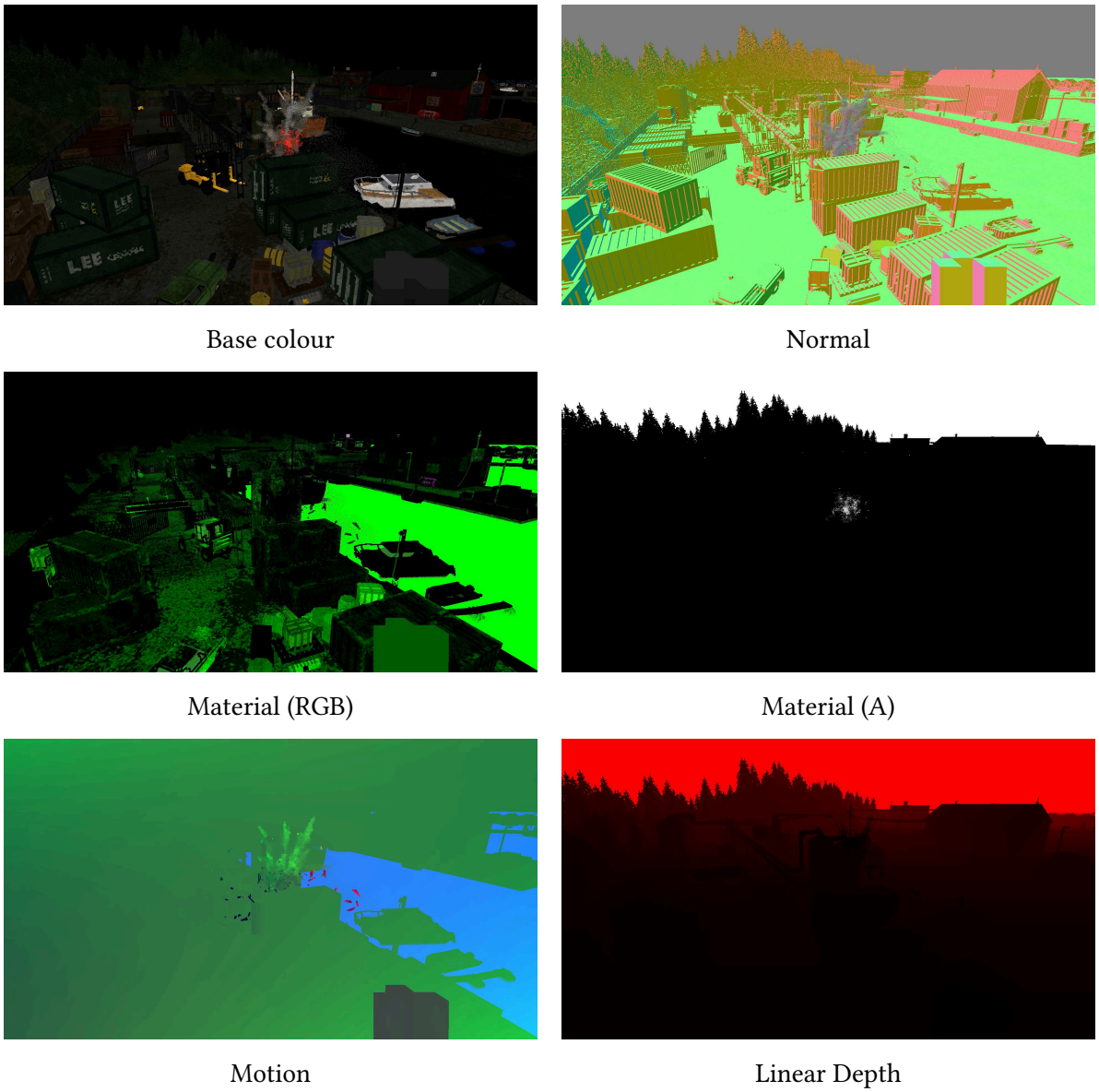


Figure 19: G-buffer of the frame in Figure 18 of *Teardown*. (Tuxedo Labs, 2022; Wittens, 2023)

To draw the voxel objects, the game draws the geometry of a box. Then, in the fragment shader , a DDA is performed through a 3D texture. To speed it up, the renderer uses 2 mipmaps to be able to skip empty space faster. The rendering is done front-to-back to avoid overdraw by checking against the linear depth buffer. The observer might have noticed that in Table 1 there are two depth buffers (called Depth and Z-buffer). This is because OpenGL cannot know the depth of each pixel, since they all lie inside the box used to render each object. To work around this problem, *Teardown* writes the depth of the pixels in the linear depth buffer and occasionally copies it to the Z-buffer.

Using deferred rendering is known to pose a problem for transparency. It is often needed to first render opaque geometry using deferred rendering and then the transparent geometry in a front-to-back order using forward rendering (Magnerfelt, 2012). To simulate 50% transparency, *Teardown* displays such materials in a chequerboard pattern and other percentages use a blue-noise texture. This technique is called screen-door transparency (Mulder, Groen and van Wijk, 1998).

After the voxels are drawn, there are two more passes. One for the various wires, which are drawn using “normal” geometry, and then the smoke and fire particles. Since these are drawn using 2D textures, they alternate their normals for each pixel to simulate depth and also use screen-door trans-

parency. Once the draw calls are done, the renderer applies a blur on the normal buffer to smooth out the edges of the voxels.

Now that the G-buffer is ready, the lighting is computed. This is done in three steps:

1. **Ambient lighting**: Computes ambient occlusion of light
2. **Sun lighting**: Computes the light coming from the sun
3. **Local lighting**: Computes the light coming from objects (lamps, fire, etc.)

All of this is computed using ray tracing. To do that, a 3D texture of the whole level is rendered by a very optimized and multithreaded procedure on the CPU. For the Marina level, it is stored in a $1752 \times 100 \times 1500$ texture that is 262 MB in size. To keep the size of it relatively small, it is stored as a 1-bit map, where each 8-bit texel stores 2^3 voxels. Two mipmap levels are added to the texture to accelerate the ray tracing.

To render ambient lighting, two rays per-pixel are cast in a random hemisphere around each point. This uses what the renderer calls "super sparse" tracing mode. This means that instead of using the fast voxel traversal algorithm by Amanatides and Woo (1987) (DDA), it will sample voxels at regular intervals. But *Teardown* also has a "sparse" tracing mode, where it will sample voxels at regular intervals. Where they differ is that the "super sparse" mode will first start at the lowest mipmap level and then progressively go up. This is illustrated in Figure 20. Whether these rays hit anything will determine the ambient lighting of the pixel.

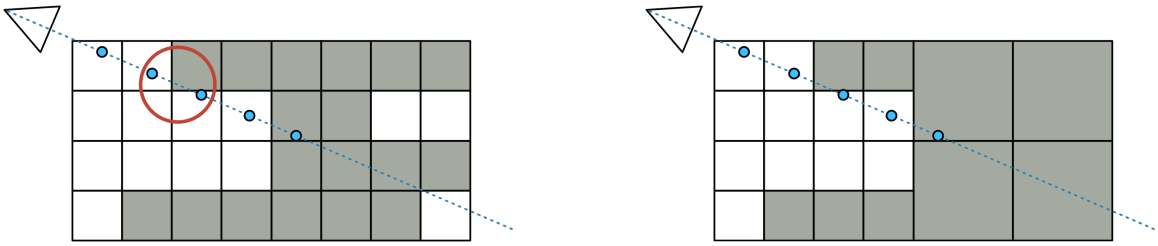


Figure 20: Illustration of the "sparse" tracing mode (on the left) and of the "super sparse" tracing mode (on the right) of *Teardown*. You can see that voxels can be missed when using this method as highlighted by the red circle. (Wittens, 2023)

Once this is done, the sun lighting is computed. For each pixel, a ray is shot towards the sun using the "sparse" method. If it hits something, the pixel is in shadow and if it hits nothing, it is illuminated. Local lights are then computed using geometry that covers the area on the screen that will impact lighting. Visibility of those is computed by tracing a ray from the pixel to a random point on the light's surface to see if there is anything in between. When doing these calculations, only the irradiance is computed, meaning the colour is not taken into account. This is because the buffer can then be denoised to hide artefacts that appear from the stochastic nature of this method.

Now that the lighting is rendered and denoised, it is applied to the base colour buffer . Then, volumetric lighting is rendered. To do that, a ray is marched from the camera and at each step, another ray is cast to check if the current light source is visible, if yes, its colour is added to the texture. Since this effect is expensive to compute, it is done at quarter resolution. This texture is also denoised.

The final step that will be explained here is reflections. If the pixel's reflectivity is above zero, a ray is shot in the direction of the reflection. The roughness value of the pixel will change the position of this ray to simulate blurry reflections. The ray will check for collisions for 32 units, if nothing is hit, it will use the skybox colour. Otherwise, it will either use the colour of the voxel if it is on screen or have a

black reflection. This buffer is then denoised and applied to the final texture which will then receive some post-processing effects.

This will not be explained in depth in this bachelor thesis, but we can highlight that one of these is Temporal Anti-Aliasing (TAA) and a side effect of this process is that it can act as a final denoising step. Other post-processing effects include:

- 2D sprites with transparency
- bloom
- depth of field
- motion blur
- automatic exposure
- colour correction
- tone mapping
- gamma correction

3.1.2 GVOX Engine

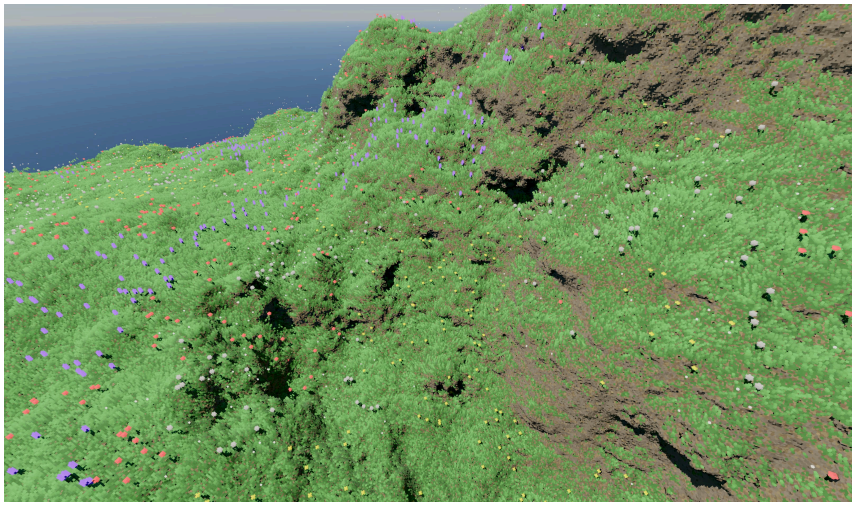


Figure 21: Screenshot of *GVOX Engine*.

GVOX Engine is an open-source project started in 2022 by Gabe Rundlett. The progress of the project is shared on Rundlett's *YouTube* channel as well as his *Discord* server. It is a fully custom voxel engine built on top of an abstraction layer over Vulkan called *Daxa*, which Gabe Rundlett also works on. *Daxa* is made for GPU-driven rendering and has a fully bindless API. This means that *GVOX Engine* is GPU-driven and almost everything related to rendering is done in compute shaders. *Daxa* also provides other utilities, such as an ImGui integration, FSR2 that can upscale the rendered image while also applying some anti-aliasing and a task graph that allows to handle synchronization with the GPU for the user (Rundlett, 2022).

GVOX Engine is an interesting project to analyse because it uses ray-tracing for the voxel rendering with global illumination. It is also using compute shaders for the rendering, which is closer to what the media project of this thesis will do compared to *Teardown*. Furthermore, the world is fully editable with good update time.

Although *GVOX Engine* is open-source, its codebase is quite large and the fact that it uses *Daxa* instead of Vulkan directly makes it more complex to understand. This analysis will try to be as complete as possible, however it will mostly focus on the information that can be gathered from tools such as *RenderDoc* and occasionally looking at the code when needed. Also, as of may 2024, this project has

switched to using hardware ray-tracing, the analysis will be performed on a previous version that was still using a compute shader.

The first rendering step is the background, which uses an atmosphere simulation algorithm implemented by Sakmary (2023). This is done in a separate command than the rest of the rendering. The voxel rendering command task graph starts.

The voxel rendering compute shaders can be broken down into 6 sections:

1. World simulation and updating
2. Primary rays computation
3. Particles rasterisation
4. Secondary rays (indirect lighting) computation
5. FSR 2
6. Blur

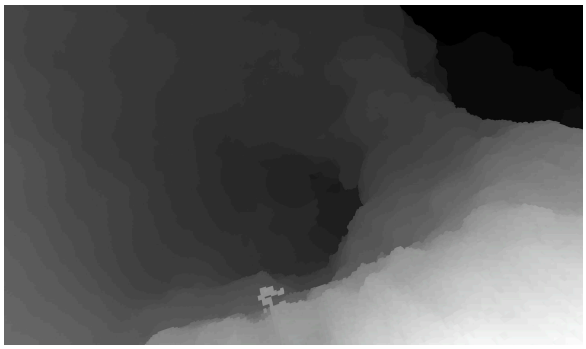
The voxel data is stored in a palette-compressed single level brickmap. Although the details may differ, the concept is probably similar as the technique presented by Hjerpbakk (2022). The idea of palette-compression in the context of voxel data, is to try to encode the presence of a voxel in as little data as possible, and store the colour and material information elsewhere, in a palette. This allows to reuse these colour data, that are often relatively big. A naive way of storing colour would be using 3 floats, which is 12 bytes for examples. In a worst case scenario where each voxel has a different colour, this way of storing data is more costly. However, as the data gets bigger and colours start to repeat, it quickly becomes worthwhile. As a palette is a dynamic structure, memory allocation should be done carefully. Furthermore, the indirection caused by the palette could negatively impact memory caches, so the data structure should be designed around it (Lars, 2023).

As mentioned above, in the first part of the rendering, the world is simulated and updated in a compute shaders. This includes moving objects such as grass or water simulation. It also includes updating the world using brushes that allow to destroy or place voxels. After that, primary rays are shot. From this, 4 buffers are filled. First, the depth buffer that is stored in `R32_FLOAT` format. Then a velocity buffer, in `R16G16B16A16_FLOAT` format. However, it remains dark most of the time due to the limited number of moving objects. Normals are stored in `R10G10B10A2_UNORM` format. Finally, a buffer called G-buffer internally that is stored in `R32G32B32A32_UINT` format is filled. They can all be seen in Figure 22.

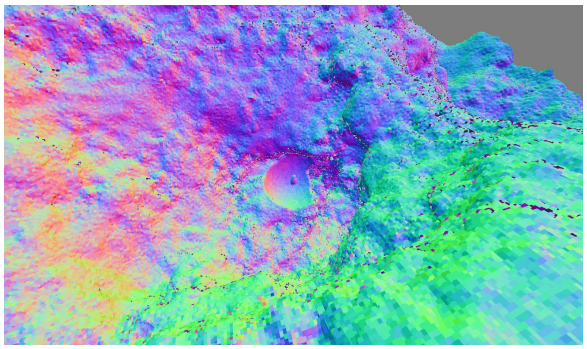
From what can be seen in *RenderDoc*, it is not exactly clear what is stored in the G-buffer and in what layout, since only the red and blue channels have data in them. But Rundlett shared on *Discord* that he was inspired by *kajiya* which is an "Experimental real-time global illumination renderer made with Rust and Vulkan" (EmbarkStudios, 2020). This is the layout explained in their documentation:

- Albedo (8:8:8, with one byte to spare)
- Normal (11:10:11)
- Roughness & metalness (2xf16; could be packed more)
- Emissive (shared-exponent rgb9e5)

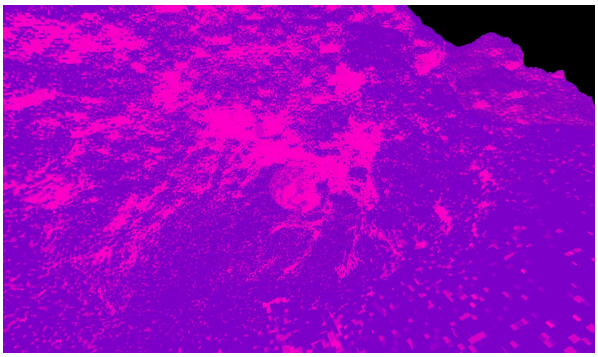
As the green and alpha channels are empty (which makes sense since normals are stored in a separated buffer), we can assume that the G-buffer stores the base colour in the red channel and the roughness and metalness in the blue channel.



Depth



Normals



G-buffer

Figure 22: Buffers filled by the primary rays in *GVOX Engine*.

One thing to note about normals, is that they are not computed based on the faces on each voxel, but they are the same on the whole voxel. This is an artistic choice to give a smoother feel to the lighting of the scene.

Now that the primary rays are cast, particles are rasterised. This includes things like grass, flowers, fire, water, etc. They are rasterised and not ray traced to improve rendering performance. As rasterisation does not have all the same effects that raytracing has, particles all have a second pass to render shadows.

Secondary rays are then cast to compute global illumination. Global illumination is computed using a technique called ReSTIR which uses resampling techniques for high-quality results (Bitterly *et al.*, 2020).

To improve performance, AMD FidelityFX™ Super Resolution 2 (FSR 2) is applied. This is a technology developed by AMD to upscale images while applying anti-aliasing (AMD, 2022). Contrary to NVIDIA DLSS 2.0, which only works on NVIDIA GPUs , FSR 2 works on any GPU . This means that for example, on a 4K monitor, the image could be first rendered at a 2K resolution and then upscaled to 4K using FSR 2.

Finally, bloom, tone mapping , gamma correction and other post processing effects are applied. Bloom makes it so that brightly lit objects on the scene have a glow around them, giving a better feel on how things are lit.

3.2 Interviews

To help guide the project and to have accurate informations on current solutions, authors of notable voxel projects where interviewed.

3.2.1 Questionnaire and interviewees presentation

3.2.1.1 Questionnaire

A questionnaire of eight questions was prepared to understand various aspect of voxel rendering. These questions where either asked on a call, or sent by email to be answered. These questions are:

1. What lead you to start working with voxels?
2. For you, what is the hardest part when working with voxels?
3. From your personal experience, what do you think is the future of voxel rendering techniques?
4. Based on what you've already done, what would be your suggestions to improve current voxel rendering?
5. Have you used HWRT? If yes, what is your personal experience with it related to voxels?
6. If you had to restart your project from scratch, is there anything that you would change?
7. What is your personal experience with voxel animations?
8. Is there anything we did not mention on the subject of voxel rendering that you think would be worth mentioning?

This questionnaire was asked to four persons that each have notable voxel experience.

3.2.1.2 Felix "Xima" Maier

Felix Maier is a graphics programmer from Germany and the author of the *VoxelChain* engine which is built using WebGPU and runs in the browser (Maier, no date). The engine features global illumination computed using cellular automata and ray tracing for reflection and refraction. It also has entity rendering with animations and ray-traced sound propagation. Before that, Maier worked on adding HWRT to the WebGPU implementation of Chromium (Maier, 2020).

3.2.1.3 Daniel "frozein" Elwell

Daniel Elwell is a graphics and game programmer from the USA who worked on a voxel engine called *DoonEngine* (Elwell, 2022). He then used the engine to create a sandbox game called *Terra Toy* (Elwell, 2023). This engine uses OpenGL and has per-voxel reflections. After *Terra Toy*, Elwell started working on a new engine that uses Vulkan and HWRT. He is currently working on it, he showed in a recent video that he added the option to have multiple kinds of data structures in the BLASes.

3.2.1.4 Ethan Gore

Ethan Gore is the author of the appropriately named *Ethan Gore's Voxel Project*. This project uses OpenGL to render a ton of voxels using rasterisation. It also allows to edit big part of the scene at once in very little time. A notable feature is the ability to import *Minecraft* maps where each texels in the texture of the voxels that are in the map are imported as one voxel. Meaning that the map has a resolution that's sixteen times bigger than *Minecraft*. This scale is achieved though a powerful LOD system and the fact that rasterisation is fast on GPUs.

3.2.1.5 Gabe Rundlett

Gabe Rundlett is the creator of the *GVox Engine*, which is an open-source voxel engine that features ray-traced global illumination and an editable terrain. Recently, he added HWRT to the engine. He also developed the `.gvox` format with Douglas Dwyer. This format allows to store voxels with multiple kinds of internal data-structure for faster loading. He was recently hired by *Tuxedo Labs*, creators of *Teardown*.

3.2.2 Results analysis

A common cause of people starting their own voxel project are other voxel games such as *Minecraft* or *Cube World*. Furthermore, the dynamic nature of voxel worlds is attractive.

The most cited answer about the difficulty of working with voxels is memory consumption. This leads to the usage of data structures that can compress memory, especially if they have sparse data, but this slows down updating speeds. The respondents agree that the trade off between memory consumption and update speed should be carefully considered. Some users expect Global Illumination from voxel games, making project potentially more difficult to make. Also, the big number of techniques available in the field can cause analysis paralysis.

The hope is for future techniques to have worlds that are even more dynamic. The interviewees concur that brickmaps offer a good balance of memory consumption and update speed. One path that should be explored is using rasterisation to speed up primary rays.

As for current techniques, there is no consensus on whether storage could be improved or not. However it is possible that respondents are not speaking of the same metric. Denoising and upscaling could be made in a way to profit from the simple shapes of cubes.

The experts don't agree on if HWRT can be faster than pure compute. The "black box" nature of the TLAS, that is not designed for voxel data, is not optimal and one written by hand could be faster. When doing HWRT, passing only surface data to the GPU can help. HWRT is designed for the divergent nature of ray tracing, meaning that it can use all of the threads of the GPU more efficiently. The recent NVIDIA GPUs that use the Lovelace architecture[1] can profit from shader execution reordering that could improve performance even more. DirectX 12 also recently got the possibility to use Work Graphs, which allows to start work on the GPU from the GPU , which could potentially avoid the problem of having idle threads like when using pure compute. However, a test was made to traverse a BVH with them, and it was slower than HWRT.

As an advice for starting a new project having the possibility of rapid prototyping was often mentioned. One way of doing it that was shared is to have a second branch that is simpler and allows to try new things. Also, big modules should be separated, allowing to change them without breaking an other part of the program.

The respondents agreed that voxel animation is a very hard problem, even more than voxel storage or ray tracing. One strategy that was cited is to voxelize a mesh model or a distance field, which is a grid or mathematical representation where each point stores the shortest distance to a surface or object within a given space.

3.3 Summary

In this section, two voxel projects where analysed in depth, *Teardown* and *GVOX Engine*. Then, four notable authors of voxel project where interviewed.

From this, we learned that the raster pipeline can be used for voxel ray tracing as shown in *Teardown* and that global illumination could be achieved with the compute pipeline using brickmaps.

During the interviews, we learned that brickmaps are an interesting option to store voxels and that there is no consensus on if HWRT can be faster for voxels than compute. Fast iteration time should also be a priority.

[1] The RTX 4000 series for example.

4 Test Protocol

4.1 Definition of Used Metrics

4.1.1 Memory Usage

As mentioned in the interviews, memory usage is the most difficult part when handling voxels, meaning that measuring this value is important. Memory access is considerably slower than computation, meaning that keeping data small can improve the cache efficiency of the program and in turn give a smoother application. The measurements will be done on scenes of varying size to see how it scales.

4.1.2 Voxel Count

Although it is directly related to the memory consumption, the amount of voxels that can be displayed by the engine is an important number. As the focus on the media project was more on the path tracing side, the data structure to store voxels is not very complex. This means that the number will not be very high compared to what could be done using a better data structure. For example, *Teardown* can store two billion voxels in its world texture (although it lacks any colour information). But it is important to measure it to compare the value once and if improvements have been done.

4.1.3 Frame Time

The goal of the media project is to be an interactive application. Meaning that the path tracing code should be able to run in real time. This can be tested by measuring the time that a frame takes to render. The number of frames per second is not used as it can be a misleading metric when doing comparisons, although it is the reciprocal of seconds per frame.

The minimum for interactive applications is often placed at $33.\overline{3}$ ms (or 30 fps), although going a bit lower is possible, it will feel considerably worse. $16.\overline{6}$ ms (or 60 fps) is considered a smooth target nowadays, although the advent of screens with high refresh rates can lead to even lower numbers.

The amount of work the GPU has to do can vary depending on the scene and the camera position, meaning that this measurements will be done with multiple camera position.

4.2 Measurement Tools

4.2.1 Custom Tools

As the media project is not made in an off the shelf engine, some of these tools will have to be custom made. One of these will be a frame time dumper, that will take the frame time value on 128 frames and write them to a file, that will then be used to compute values (average, median or other).

4.2.2 NVIDIA Nsight Graphics

NVIDIA Nsight Graphics is a tool that attaches itself to an application and allows to measure its performance. It has tools such as GPU Trace and Trace Analysis that can help find hotspots and the percentage of frame time of shaders. It can also give explanations on performance issues. This tool is only available for NVIDIA GPUs , meaning that another one must be used of AMD GPUs .

4.2.3 AMD Radeon GPU Profiler

AMG Radeon GPU Profiler is an equivalent to Nsight Graphics for AMD GPUs , with a stronger focus on profiling, so it lacks the debugging features. It will be helpful to profile the application on the AMD hardware where test will be run.

4.3 Measurement Platforms

To measure these numbers, two computers will be used. One has an NVIDIA graphics card and the other has an AMD one. The first one is a laptop and the other one a desktop.

4.3.1 AMD Desktop

OS	Windows 11 Pro (10.0.22631)
CPU	AMD Ryzen 9 7900X3D, 4401 Mhz, 12 Cores
GPU	AMD Radeon RX 7900 XT
GPU Driver Version	31.0.24033.1003
RAM	32 GB

Table 2: Hardware specifications of the desktop.

4.3.2 NVIDIA Laptop

This laptop is a Lenovo Legion Slim 7 Gen 8 (16″ AMD). During the tests, it will be set to its maximum power profile.

OS	Windows 11 Pro (10.0.22631)
CPU	AMD Ryzen 7 7840HS, 3801 Mhz, 8 Cores
GPU	NVIDIA GeForce RTX 4060 Laptop
GPU Driver Version	552.44
RAM	32 GB

Table 3: Hardware specifications of the laptop.

4.4 Project Setup

To make reliable measurements, the app has a benchmark mode where the camera is fixed and ImGui is disabled. The measurements will be done on two camera placements, once with per-voxel lighting and once with per-pixel lighting. This will be explained in more details in Section 5. To test if the technique called “russian roulette”[2] speeds up the rendering, it will be done on one of the camera positions and compared on both computers using Student’s t-test.

The first placement, called “Bench 1”, faces a wall and the rays don’t have to travel a lot of empty space compared to the second one, called “Bench 2”, which is in a corner (see Figure 23). Bench 2 can also see the reflective ground which leads to more bounces. This could mean that Bench 2 might have a higher frame time.

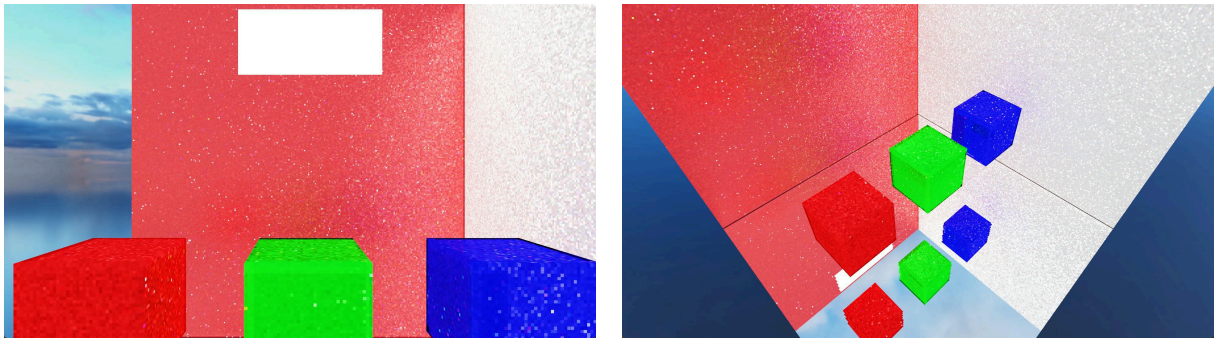


Figure 23: The two camera placements for the benchmarks. On the left “Bench 1” and on the right “Bench 2”.

[2] “Russian roulette is a technique that can improve the efficiency of Monte Carlo estimates by skipping the evaluation of samples that would make a small contribution to the final result” (Pharr, Jakob and Humphreys, 2023, para. 2.2.4)

The measurements done using per-pixel lighting might also have worse performance, since neighbouring pixels have branches that are less coherent, which is problematic for the parallel nature of a GPU.

4.5 Summary

In this section we explained what measurements will be made and how. The measurements will be frame time, memory consumption and voxel count. As for the tools, some will be custom made and for the rest, NVIDIA Nsight Graphics and AMD Radeon GPU Profiler will be used. The program will be tested on two computer, a laptop and a desktop. This with multiple parameters to test the renderer under different conditions.

5 Media Project

This chapter will present the environment, limitations and the production of the media project.

5.1 Development Environment

The project is programmed on a Windows 11 machine. It first started as a Rust 1.78.0 project, which was developed using RustRover 2024.1 EAP by JetBrains. The project used these packages:

- anyhow
- ash
- ash-window
- log
- nalgebra
- pretty_env_logger
- thiserror
- winit
- raw-window-handle
- vk-mem
- imgui
- imgui-winit-support
- bytemuck
- nalgebra-glm

And also packages that these ones use.

Then, the project was moved to Jai beta 0.1.091. First Focus was used to edit the code, which is an editor made in Jai that has built-in syntax highlighting for the language. However, it does not support "go to definition" and other convenience features, so Visual Studio Code was used with the Jails plugin afterwards.

Git is used for version control, and the repository is hosted on GitLab.

5.2 Limitations

The goal of the project was in part to implement it using Vulkan, meaning that a significant amount of the allowed time was spent on that. Vulkan is a very verbose API that forces the programmer to write many lines to get started. The official Vulkan sample to display a triangle is 1200 lines long (Khronos Group, 2019). This took time from the voxel rendering part of the project meaning that not all of the goals where achieved. The most important missing features are voxel models loading from `.vox` files and storage compression using Brickmaps.

Another limitation is the visual results achieved by the path tracing. Although it is path tracing, it is very noisy and the accumulation is restarted every time the camera moves. These are points that should be fixed to be able to use the renderer in a final product.

5.3 Production

5.3.1 Rust and Vulkan Tutorials

Before working on voxel rendering, a Vulkan project has to be setup as a base. Since at the time Rust was the planned programming language, a decision had to be made on the Vulkan wrapper that would be used. There are three main options available: (1) Vulkano, which is a relatively high-level and safe wrapper. (2) Ash, a lower-level wrapper that still provide some convenience by, for example, returning

`Vec<T>` instead of slices. (3) Vulkanalia, also a lower-level wrapper that is heavily inspired by Ash. The key difference is that Vulkanalia has a very clear distinctions between the Vulkan bindings in the vulkanalia-sys crate and the wrapper in the vulkanalia crate. Since the goal is to be able to distinguish when Vulkan is used versus the bindings, this is a good thing.

So the choice was made to use Vulkanalia. The plan was to follow a tutorial named "Vulkan Tutorial" (Overvoorde, 2023) and conveniently, a version using Vulkanalia exists (Mayes, 2020). Finishing this tutorial lead to the image in Figure 24. The Rust project has seventy five dependencies and sub-dependencies. The total number of lines of code is roughly two thousands.

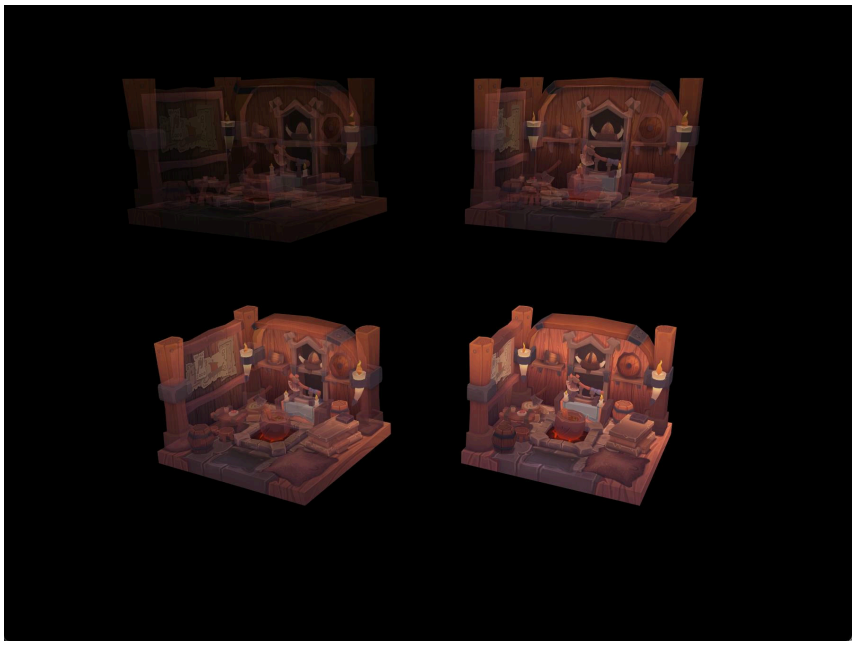


Figure 24: Screenshot of a frame rendered using the Vulkan renderer made with Vulkan Tutorial.

Once this was done, another tutorial was followed called "VulkanGuide" (vblanco20-1, 2020). This tutorial goes further than Vulkan Tutorial by, for example, starting out with compute pipeline before a triangle one. It also uses more libraries such as vk-bootstrap, Vulkan Memory Allocator (VMA) and ImGui.

The library vk-bootstrap does not have a Rust equivalent, but most of the benefits it brings could be replaced by functions made in Vulkan Tutorial. However, when it came to using VMA, there were two options. Either vk-mem, which is a wrapper around VMA, or gpu-allocator, which is a custom allocator made directly in Rust. The problem is that both of those do not use the types brough by Vulkanalia, but the ones from Ash. Meaning that it is not possible to use them with Vulkanalia. At this point, the choice was made to transition the project from Vulkanalia to Ash to be able to use one of these allocators.

Now that the project used Ash, an attempt with the gpu-allocator library was made since it seemed to be the standard for Rust projects. But it was changed to vk-mem as it did not have separate functions to allocate images and buffers, and the goal was to avoid issues from anything that could go wrong from this.

Then, ImGui needed to be added. Conveniently, a crate called imgui exists. However, it does not wrap the backends that come with the C++ version. They have their own backends, but they don't include a Vulkan one. So to be able to render with Vulkan, the imgui-rs-vulkan-renderer crate was imported to the project. The library requires to pass in the vk-mem allocator wrapped in a mutex in an atomically

reference counted type. To be able to free the allocator at the correct time, it was also wrapped in a ManuallyDrop: `gpu_allocator: ManuallyDrop<Arc<Mutex<vk_mem::Allocator>>>`.

ImGui could now be rendered, but it still needed to respond to window events, which was managed by the crate winit. To do this, the crate imgui-winit-support exists in the imgui project, however at the time, it was using winit 0.27 and the tutorial project used winit 0.29. To circumvent this, the backend was forked and ported to use winit 0.29.

The passing of window handles in Rust is usually done via the raw-window-handle crate. It has a common type that windowing libraries can use to so that other libraries that would want to use a window don't have to rely on a specific windowing library. However, not every crate depend on the same version of raw-window-handle, like for example imgui-winit-support. So I had to enable a feature in winit to ask it to use version 0.5 of raw-window-handle.

The VulkanGuide tutorial was not completed as not every feature presented were necessary for voxel rendering. ImGui and compute shaders were the most important ones. The base of the triangle pipeline was still implemented.

The image in Figure 25 is the one that the project had when the tutorial was stopped. In the middle you have Blender's test model Suzanne that is loaded from a GLTF 3D model file. Its texture is a generated chequerboard. The window on the top left is an ImGui window. The render scale allows to change the virtual resolution of the image, which is then upscaled on the window. The effect index allows to chose which compute shader to use for the background. Data 1 through 4 are push constants parameters that the compute shaders use. The first background is a gradient between colours provided in Data 1 and 2. The second background is a sky with stars, with a background colour controlled by Data 1.

This project has approximately 3'300 lines of code and 159 dependencies.

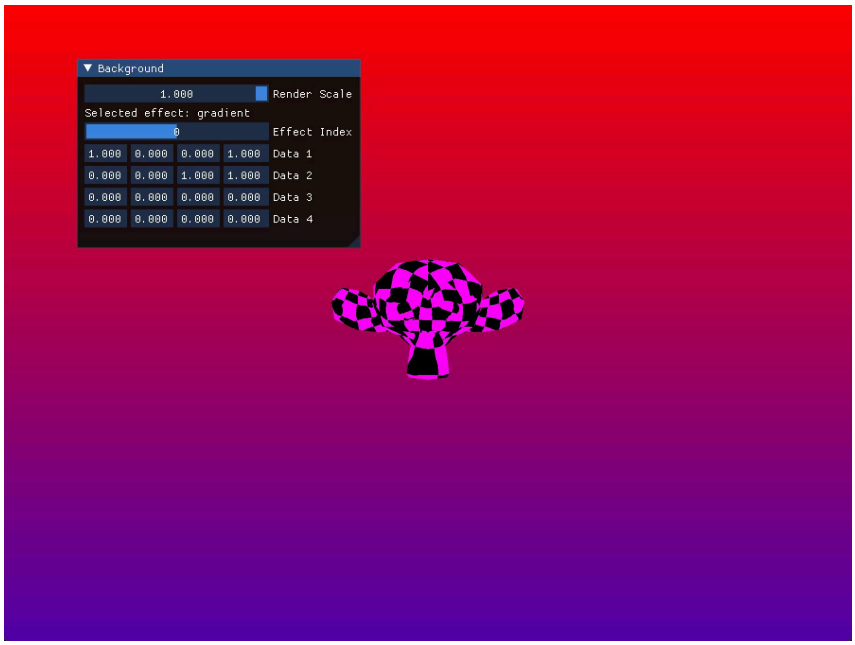


Figure 25: Screenshot of a frame rendered using the Vulkan renderer made with VulkanGuide.

To start the real voxel renderer project, the tutorial one was copied to a new folder. First, the mesh pipeline and GLTF loading was removed. Then, dependencies were updated one by one. winit changed its API , requiring to change the structure of the project significantly. The crate imgui-winit-support was upgraded, meaning that raw-window-handle 0.6 was able to be used, but it still required a fork. The crate vk-mem was upgraded to 0.4 and Ash to 0.38, but this means that imgui-rs-vulkan-renderer

could not be used any more since it requires vk-mem 0.3 and Ash 0.37. Now the choice was either write a custom ImGui renderer or downgrade every dependencies making all the previous work useless.

5.3.2 Jai and Vulkan Guide

At the same time, access to the private beta of the Jai programming language was granted. As the dependency situation was a bit complicated with Rust, Jai was tried and then chosen for the media project. The Jai compiler comes with a how_to folder which is a tutorial with code examples. It also comes with a modules folder which contains the standard library of the language alongside bindings for various libraries such as SDL, ImGui and Vulkan. A module in Jai is the word used for libraries, to not mix things up with C++ libraries[3].

The objective was to get back to the same point as the Rust project, by having a compute pipeline and ImGui working. First, a window was opened using SDL which was easy enough. Then using Vulkan with the provided module worked well, until Vulkan 1.3 was required. This module only provides Vulkan 1.0, so a project local module was created to update it. Jai comes with a Bindings_Generator module that is used to generate the Vulkan bindings. A `generate.jai` file is present in the project that reads the .h files and generates `.jai` files. So Vulkan 1.3 headers were provided to the generator and some tweaking was made with the function that translates name when handling numbers. These updated bindings where uploaded to their repository[4].

Then VMA was added to the project. Jai does not have VMA bindings by default, but Kofu on the Jai Discord posted their bindings. These where then updated, reorganized a bit and also put in their own repository[5].

With all of these assembled, a demo project using SDL and a Vulkan compute pipeline could be made. The modules presented earlier where included in the project using Git Submodules. This demo project was put in a repository as an example[6].

The next step was to add ImGui. Jai comes with ImGui bindings, but it is not its latest version, and is not the one from the docking branch, that adds more features. So it was updated and put into a separate repository[7]. But this was not enough, as just like in Rust, the ImGui bindings don't have the backends. The next step was then to translate the C++ backend into Jai manually, which is almost two thousand lines of code for the SDL2 and Vulkan backends. After this was done, since it caught up to the Rust project and has the code you would have at the end of chapter two of VulkanGuide, it was also put in a repository[8].

5.3.3 Build Setup in Jai

In C++, to define a complete program, you cannot only rely on .cpp files. Some of it is defined through the way the program is compiled, through flags passed to the compiler. To make this easier, third party tools like scripts, Makefiles or CMake are used. But this requires to learn a new scripting language that are often quite complicated.

In Rust, the compilation is helped by a tool shipped with the compiler: Cargo. The compilation is defined in a Cargo.toml file that can also have a list of dependencies. This makes it easier than in C++, as it is a standard way that every project uses. That being said, it is still limited to what Cargo allows you to enter in this file.

[3] This was probably decided before C++20 modules
[4] https://gitlab.com/Stowy/jai-vulkan
[5] https://gitlab.com/Stowy/jai-vma
[6] https://gitlab.com/Stowy/jai-vulkan-sdl
[7] https://gitlab.com/Stowy/jai-imgui
[8] https://github.com/St0wy/Jai-Vulkan-Guide

To help with these problems, Jai uses the approach of defining the build in a Jai file, similarly to what Zig does. Then, building the project is only a matter of running the command `jai build.jai`. This method has the advantage of being able to share code between the build code and the runtime code.

In the media project, the build was setup as so: first, arguments are read to see if we want to do a debug, optimized or release build. Then the shaders are compiled by invoking glslc. After that, a build directory is created for the executable and compilation flags are set. Finally, the build is performed. The code looks like that:

```jai
// #run tells the compiler to execute the code at compile time
#run Build();
Build:: () {
  w:= compiler_create_workspace();
  target_options:= get_build_options(w);
  args:= target_options.compile_time_command_line;

  build_optimized:= false;
  build_release:= false;

  for arg: args {
    if arg == {
      case "optimized";
        build_optimized = true;
      case "release";
        build_release = true;
    }
  }

  BuildShaders(debug = !build_optimized && !build_release);

  target_options.output_executable_name = "voxels";
  set_build_options(target_options, w);

  if build_release then BuildRelease(w);
  else if build_optimized then BuildOptimized(w);
  else BuildDebug(w);

  add_build_file("src/main.jai", w);

  set_build_options_dc(.{do_output=false});
}
```

Listing 2: Build procedure of the `build.jai` file.

The compilation of shaders is quite easy thanks to Jai's `run_command()`.

```jai
HasGlslc:: () -> bool {
  command:= break_command_into_strings("glslc --version");
  result, out, error:=
    run_command(..command, capture_and_return_output = true);
  return result.exit_code == 0;
}

BuildShaders:: (debug: bool = true) -> bool {
  if !HasGlslc() then return false;

  command:= ifx debug {
    break_command_into_strings("glslc voxel_render.comp -o
    voxel_render.comp.spv --target-env=vulkan1.3 -g -O0 -Werror");
  } else {
    break_command_into_strings("glslc voxel_render.comp -o
    voxel_render.comp.spv --target-env=vulkan1.3 -O -Werror");
  };

  result, out, error:=
    run_command(..command, "src/shaders", capture_and_return_output = true);

  if result.exit_code != 0 then return false;

  return true;
}
```

Listing 3: Procedures to compile shaders in `build.jai`.

The setup of the build directory creates the folder if it does not exists, and then copies the dll of used libraries.

```jai
SetupBuildDir:: (dir_path: string) {
  make_directory_if_it_does_not_exist("run");
  make_directory_if_it_does_not_exist(dir_path);

  #if OS == .WINDOWS {
    sdl_dll_path:= tprint("%/SDL2.dll", dir_path);
    if !file_exists(sdl_dll_path) then
      copy_file("modules/SDL/windows/SDL2.dll", sdl_dll_path);

    vma_dll_path:= tprint("%/VMA.dll", dir_path);
    if !file_exists(vma_dll_path) then
      copy_file("modules/jai-vma/windows/VMA.dll", vma_dll_path);
  }
}
```

Listing 4: Procedures to setup the build directory in `build.jai`.

5.3.4 Voxel Ray Tracing

To start rendering voxel, they had to be passed to the compute shader. This was done by passing a `VkDeviceAddress` via push constants. The material model was pretty simple, voxels are each `u32` with one byte to indicate the material and three for the colours.

To test the rendering, a world was generated using perlin noise. The code was translated to Jai from the voxpopuli template for the "Ray Tracing with Voxels in C++ Series" blog posts (Bikker, 2024b). A lot of techniques implemented in the rest of this chapter are based on this series.

Once the voxel were accessible in the compute shader, it was time to render them. The first step was to shoot primary rays and to display the voxel's normals. The traversal of the volume was done using DDA, in a branchless manner (fb39ca4, 2013). The results of this can be seen in Figure 26.

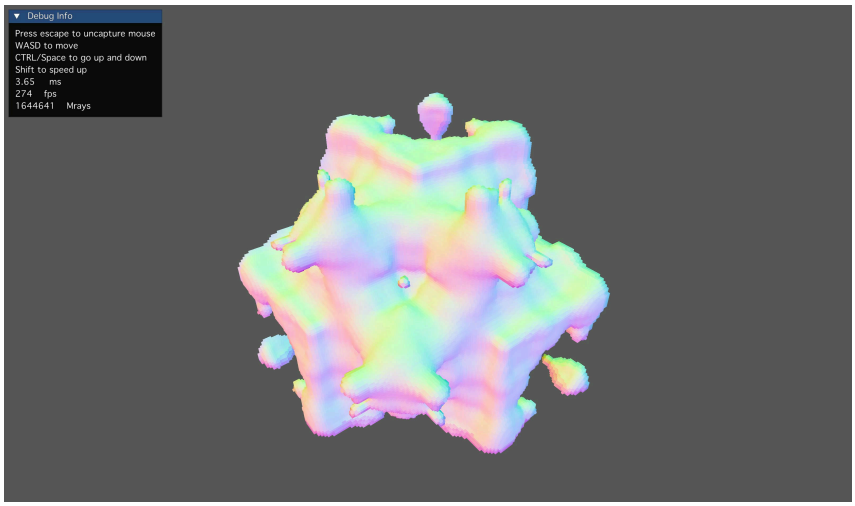


Figure 26: Voxels rendered using the DDA algorithm.

The generation of the world was really slow in debug mode, since it does not use LLVM to speed up build time. It took upwards of 40 seconds to generate. To remedy that, the library osor_cpu_dispatch was imported to have an easy way to do parallel fors. This cut the generation time by ten.

After that, ray traced shadows were added alongside Phong shading. The idea is to shoot a ray from the pixel collision point towards a light source and check if anything hits.

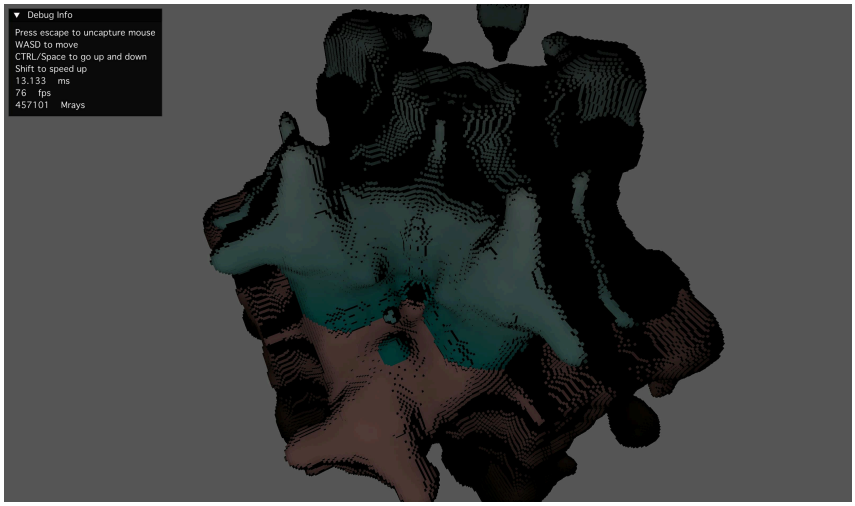


Figure 27: Ray traced shadows.

This method of doing ray traced shadows gives hard shadows that are not realistic. A method to have smooth shadows is to shoot multiple rays into a volume randomly and accumulate the result. As it

can start to be costly, from this point on, the app renders at half resolution and then upscales it on the screen. Furthermore, since the shadows are accumulated on the buffer, we can add anti-aliasing by offsetting the sample position inside the pixel at each frame. This is done using a Halton sequence that gives better coverage at low sample count. At first the sampling of soft shadows was done using white noise. To have better result at lower sample count and to improve denoising capability, it was changed to blue noise.

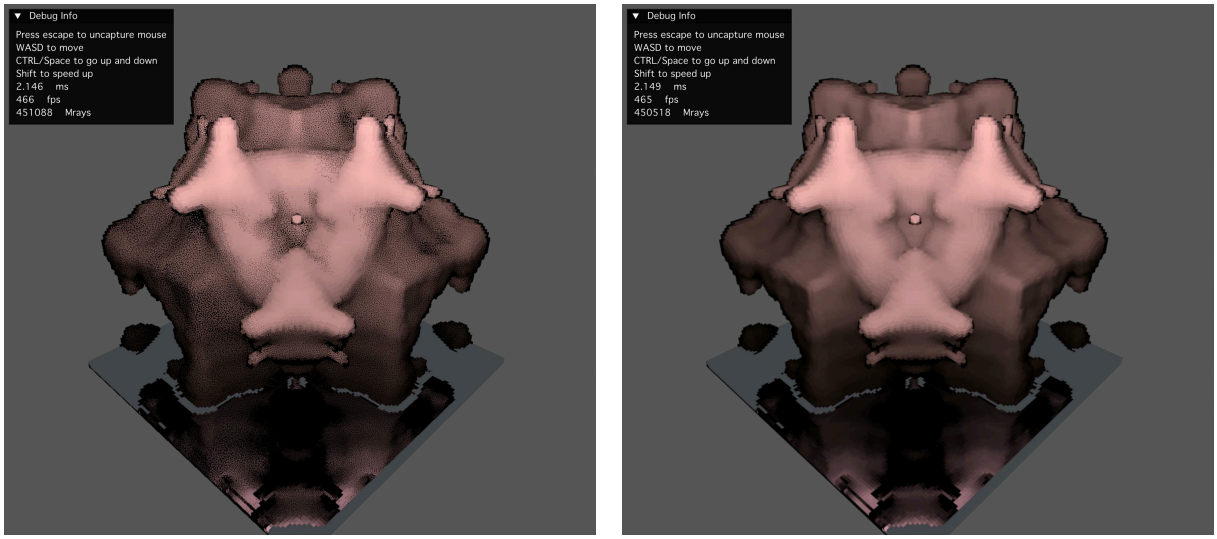


Figure 28: Smooth shadows. On the left is the first frame before accumulation, on the right is during accumulation.

World loading and saving was also added. If a world file exists, it is loaded, otherwise, the world is procedurally generated and then dumped into a file so it can be read back.

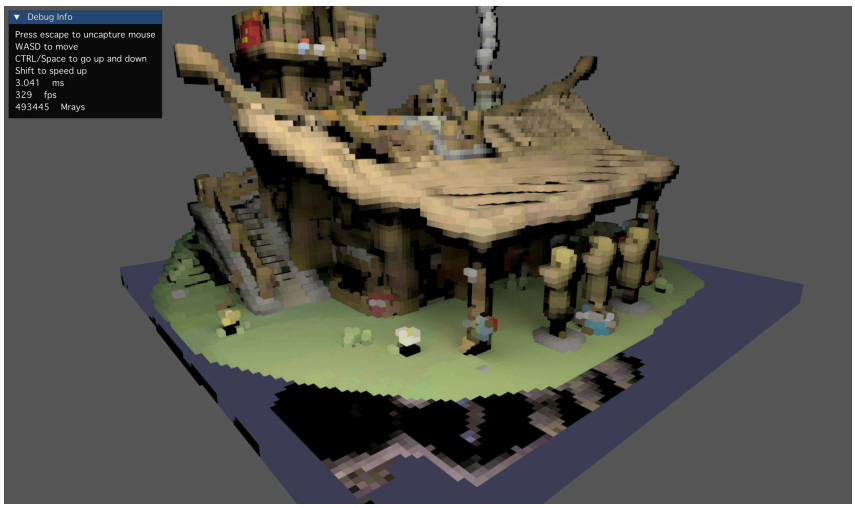


Figure 29: Example of map loading. Map created by Debusschere, P. and voxelized by Bikker, J.

Although adding soft shadows helps with the realism of the image, it still does not have indirect lighting. For this, path tracing must be used. The algorithm for it is not too complex, as explained in Listing 1. On Figure 28, we can see that the colour is not uniform on each pixels, because the seed of the random number generator is different for each of them. As an artistic choice, it was decided to seed the random number generator with the position of the voxel, making each of them have a uniform colour. This makes the first frame of accumulation less usable, but after a few frames, it looks similar to per-pixel random values.

This technique would also allow to accumulate the lighting values on a world space buffer instead of a screen space one. Thus allowing to keep the accumulation after the camera moves. It was not implemented in this project.

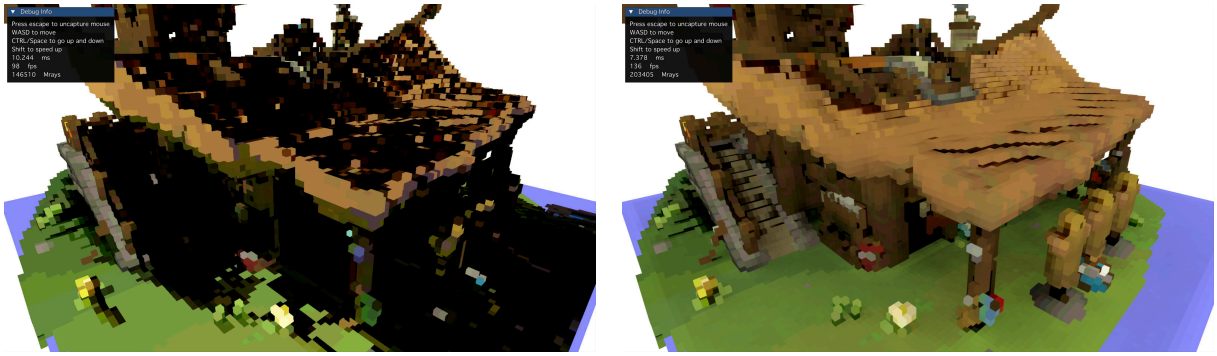


Figure 30: Per-voxel path tracing. On the left is the first frame before accumulation, on the right is during accumulation.

In this version the sky is a uniform colour. To add variety, a .hdr map was used instead. This replaces the background colour and also allows to use it as a light source. Tone mapping and gamma correction were also added.

On Figure 31 some voxels appear brighter than others because there is a sun with high values on the map, but that is over a very small area. This means that there is a very low chance of hitting the sun. One way to prevent that would be to use importance sampling on the sky, with a bias towards high intensity values, but this was not implemented.

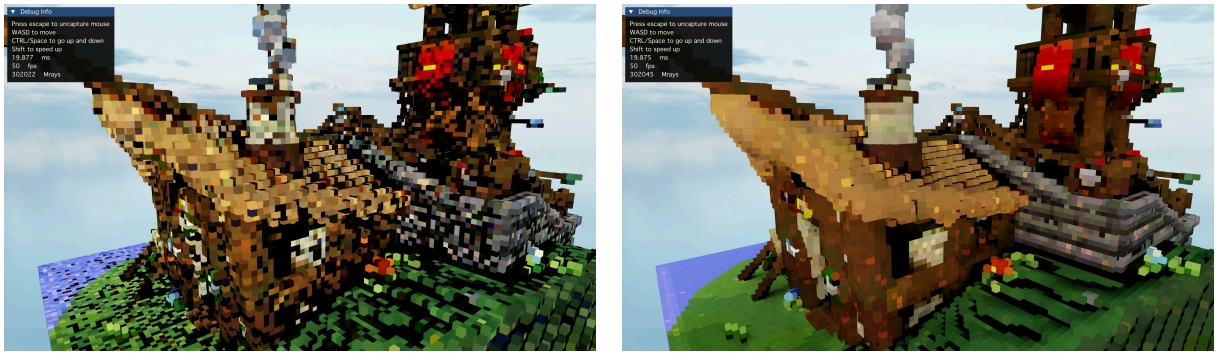


Figure 31: Path tracing with an environment map as a light source. On the left is the first frame before accumulation, on the right is during accumulation.

As mentioned earlier, the lighting and randomising is done per-voxel. But, with a few lines changed, it is possible to have per-pixel lighting. So an option was added to enable that. The amount of downscaling is also configurable between one (so no downscaling) to four. It works by dividing the image resolution by this number. It is also possible to enable russian roulette, which will randomly stop the bounce of rays that would be too long and therefore cost too much. Finally, reflections where reintroduced using a blog post by Wolfe (2020).

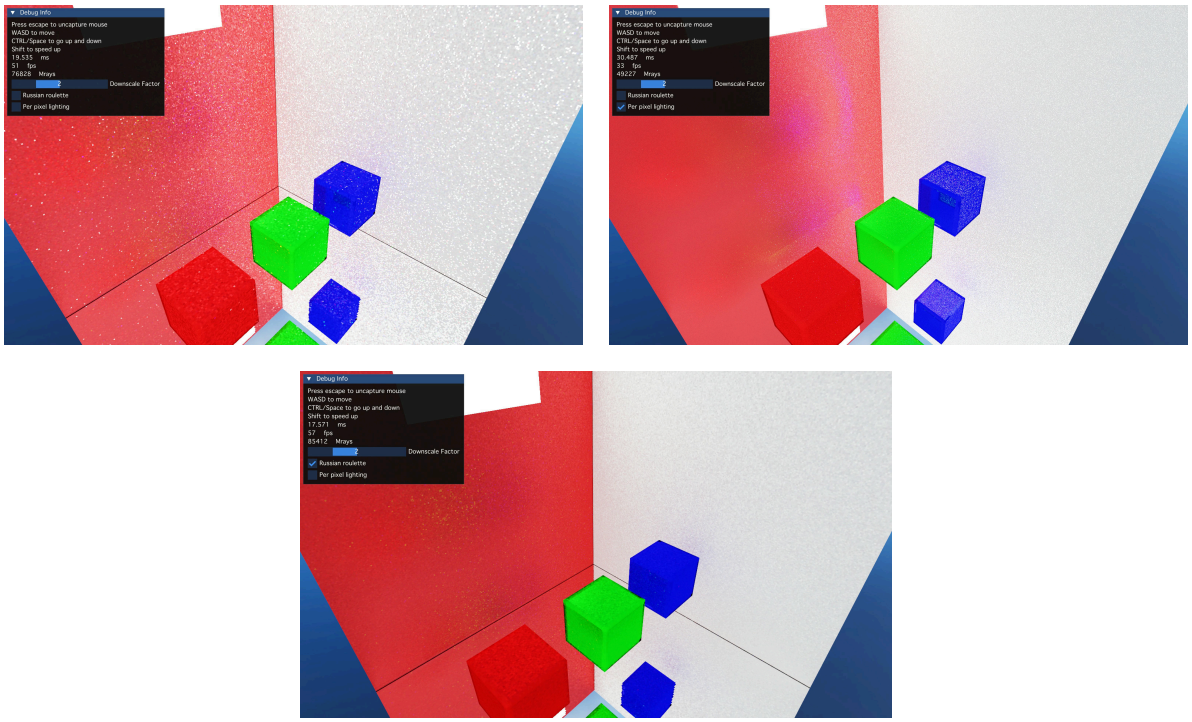


Figure 32: Final render of the project. Top left is per-voxel lighting. Top right is per-pixel lighting. Bottom has russian roulette enabled.

Up until now, the voxels were stored in a `u32`. But this limits the amount of data that can be stored per-pixel. A solution could be to use an array of structs to store the voxels, but this would significantly increase memory consumption. A solution to this is to have two buffers. One for the voxels and one for the materials. The voxels will then index into the material buffer. This technique is called palette-compression.

The size of the material index dictates the maximum size of the material buffer. For this project, a size of one byte, so 256 materials was chosen. Shaders don't support integers that are of a different size than four bytes, so the one byte indices have to be extracted. Given a voxel position, the material index is extracted using the method presented in Listing 5.

```glsl
uint Sample(ivec3 pos)
{
  int signed_index = pos.x + pos.y * GRID_SIZE + pos.z * GRID_SIZE_2;
  if (signed_index < 0 || signed_index >= GRID_SIZE_3) return 0;

  uint index = uint(signed_index);
  uint packed_index = index / 4;
  uint sub_index = index % 4;

  uint packed_material_index =
    push_constants.voxel_buffer.voxels[packed_index];
  uint material_index = (packed_material_index >> (sub_index * 8)) & 0xff;

  return material_index;
}
```

Listing 5: Code that extracts the material index from the voxel buffer. Four material indices are in one `uint`.

This material system broke the world saving and loading and would require more code to fix. Therefore, the choice was made to drop that feature and have the scene be generated by code at the start of the program every time.

The voxel material is presented in Listing 6.

```jai
VoxelMaterial:: struct
{
  base_colour: Vector3;
  roughness: float32;
  emissive: Vector3;
  ior: float32;
  normal: Vector3;
  padding: float32;
  specular_colour: Vector3;
  specular_percent: float32;
}
```

Listing 6: Material of the voxels. The padding is necessary because of GPU layout requirements. `ior` is the index of refraction.

5.4 Summary

In this chapter, the environment, limitations and production of the media project where presented. Starting from the tutorials in Rust, followed by the translation in Jai and ending with the voxel path tracing.

The repository containing the code and the windows executable from the media project can be found on GitLab[9].

[9] https://gitlab.com/Stowy/voxel-ray-tracer

6 Quantitative Analysis

This section will present the numbers measured from the media project specifically frame time, memory usage and voxel count as described in Section 4.1 using the tools listed in Section 4.2.

6.1 Memory Consumption

6.1.1 RAM

The first version of the voxel buffer used `uint` to store them. With one byte serving as the material index and then three bytes for the colour. The voxels could be either diffuse, or reflective. If they where reflective, the colour would be the tint of the reflection.

In an array of size 256^3 with each voxel being four bytes, it would take 67′108′864 bytes or 64 MiB.

But the current version of the project has a material type that is 64 bytes. In the same array, it would make it 1′073′741′824 bytes long or 1 GiB. This is why palette-compression was used, which means storing the materials in an array that is then indexed by the voxels. Each voxels are then a one byte index, that is used to fetch the material. This means that the material buffer is limited to 256 materials. In our case of having 256 materials and a 256^3 voxel buffer, the voxel buffer is 16′777′216 bytes or 16 MiB. The material buffer is 16′384 bytes long so 16 KiB. This totals to 16′400 KiB.

This is not the only allocation in the program however, measuring the total memory consumption of the program yields 145,43 MiB on the laptop and 145,23 MiB on the desktop.

6.1.2 VRAM

Although the voxel buffer is allocated in RAM, it is then transferred to the GPU and therefore copied in the VRAM. But this is not the only thing allocated, there are two `R16G16B16A16_SFLOAT` buffers for the HDR rendering and one `R8G8B8A8_UNORM` for LDR rendering. There are also two other textures for blue noise and the environment map. Finally, the swapchain also takes memory. The LDR and HDR buffers have the size of the highest resolution screen on the computer. The laptop has a 3200 x 2000 screen and the desktop a 2560 x 1440 screen.

The VRAM consumption was measured at 545′884 KiB on the desktop with an AMD graphics card and 310′384 KiB on the laptop with an NVIDIA graphics card. This difference could be cause by two things: First, each driver has its own implementation that can do different allocations for the same code. Second, it is possible that the AMD driver decided it was okay to allocate more, since the graphics card has 20 GB of VRAM compared to the 8 GB of the NVIDIA one.

6.2 Voxel Count

The voxels are stored in a fixed array of size 256^3. This allows to store 16′777′216 voxels. The fixed size could be increased, but would take up more RAM and VRAM make the traversal slower.

This static array also means that it is not be possible to have an infinite world, a more complex data structure allowing to load and unload chunks would be necessary.

6.3 Frame Time

Frame time is the most important metric of this project since the goal is to have an interactive application. This section will start by presenting the frame times under different conditions on the two computers used for measurements. Then, it will determine if russian roulette really has a performance impact.

	Bench 1 Per-voxel	Bench 1 Per-pixel	Bench 2 Per-voxel	Bench 2 Per-pixel
Average	13,82 ms	28,49 ms	15,57 ms	26,37 ms
Std. Dev	0.40 ms	1,98 ms	0,08 ms	0,76 ms
Variance	0.0002 ms	0,0039 ms	0,0000 ms	0,0006 ms

Table 4: Frame time measurements done on the AMD Desktop (see Table 2).

	Bench 1 Per-voxel	Bench 1 Per-pixel	Bench 2 Per-voxel	Bench 2 Per-pixel
Average	18,88 ms	38,18 ms	23.33 ms	37,78 ms
Std. Dev	0,29 ms	0,74 ms	0,58 ms	0,69 ms
Variance	0,0001 ms	0,0006 ms	0,0003 ms	0,0005 ms

Table 5: Frame time measurements done on the NVIDIA Laptop (see Table 3).

As guessed in Section 4.4, the second benchmark is slower. This is probably due to the higher number of bounces and the higher number of needed iterations though empty space. We can also observe that per-pixel lighting is slower than per-voxel. This is probably due to the fact the neighbouring pixels follow the same path (and therefore the same branches in the code) less often. The standard deviation increases when using per-pixel lighting, this could lead to a choppy feel during gameplay.

	With Russian Roulette AMD Desktop	Without Russian Roulette AMD Desktop
Average	11,85 ms	13,03 ms
Std. Dev	0,29 ms	0,11 ms
Variance	0,0001 ms	0,0000 ms

Table 6: Frame time of the renderer with and without russian roulette enabled on the AMD desktop. Tests done on the Bench 1 camera placement with per-voxel lighting.

	With Russian Roulette NVIDIA Laptop	Without Russian Roulette NVIDIA Laptop
Average	17,66 ms	19,85 ms
Std. Dev	0,70 ms	0,51 ms
Variance	0,0005 ms	0,0003 ms

Table 7: Frame time of the renderer with and without russian roulette enabled on the NVIDIA laptop. Tests done on the Bench 1 camera placement with per-voxel lighting.

Now, we will measure if russian roulette helps to lower the frame time. Table 6 and Table 7 show that the average frame time is higher without russian roulette, but we will need to show it statistically. To do this, we will use Student's t-test. This requires to define a null hypothesis that is the following: there is no change in frame time when using russian roulette. We also need an alpha that was defined to be 0,05. After that, the 95% confidence interval was computed to know by how much the russian roulette changed frame time.

	AMD Desktop	NVIDIA Laptop
Null hypothesis	false	false
95% C.I. Lower Bound	1,13 ms	2,04 ms
95% C.I. Upper Bound	1,23 ms	2,34 ms

Table 8: Result of the Student's t-test, with lower and upper bound of the confidence interval.

On both the AMD desktop and the NVIDIA laptop, the t-test yielded that the null hypothesis is false, meaning that there is a change in frame time when using russian roulette. The confidence interval indicates that russian roulette reduces frame time by 1,18 ± 0,1 milliseconds on the AMD desktop and by 2,19 ± 0,3 milliseconds on the NVIDIA laptop. We can observe that NVIDIA profits from russian roulette more than AMD.

7 Conclusion

In this thesis, we were able to study what voxels are, different methods to store them and what the general theory for path tracing is. The techniques studied where the Fast Voxel Traversal Algorithm by Amanatides and Woo (1987), Sparse Voxel Octrees, Sparse Voxel Directed Acyclic Graphs, brickmaps and then HWRT. We saw how different games and engine render and store voxels, such as *Minecraft*, *Cube World*, *Ethan Gore's Voxel Project* and *Octo Voxel*. Besides, people who worked on notable voxel projects were interviewed. These persons are Felix Maier, author of the *VoxelChain* engine, Daniel Elwell, author of *Terra Toy*, Ethan Gore, author of *Ethan Gore's Voxel Project* and Gabe Rundlett, author of *GVox Engine*.

We could then define a test protocol that would serve to establish what measurements would be done and how. These measurements where memory consumption, voxel count and frame time. After that, a media project was developed using Jai and Vulkan, which first started as a Rust project. We were able to see differences between the two languages and some of the difficulties that come with using Vulkan as a graphics API . Finally, we measured the performance of the renderer on two platforms and tested if russian roulette could decrease frame time or not.

7.1 Results

During Section 3.2 we interviewed experts and they agreed that brickmaps have a good trade off between memory consumption and update speed. We also learned that there is no consensus on if HWRT can speed up voxel rendering.

With roughly five thousands lines of code in Jai, a voxel renderer using Vulkan was successfully developed. Before that, difficulty with the library ecosystem of Rust was encountered. Especially with the fact the some libraries rely on specific Vulkan bindings or Vulkan memory allocator crates. The fact that the API of the crates in the project evolved quickly meant that updating them was challenging.

To store voxels, a palette-compressed array was used. This project was then measured and we could observe that memory consumption was higher in the AMD desktop. The numbers also demonstrated that per-pixel lighting has a significant impact on performance and that the position of the camera does too.

While this work sheds light on the subject and contributes to the overall understanding, it is important to consider limiting factors such as a lack of professional experience in the field and the significant amount of time required to study both Vulkan and Jai.

7.2 Future Work

Although a media project was developed for this thesis, it does not showcase more complex techniques due to time limitations.

Firstly, a better data structure could be used to store voxels, that could profit from sparse data, such as brickmaps or SVDAGs.

Then, path tracing could be improved by adding refraction and using techniques such as ReSTIR to reduce noise and denoising techniques. Additionally, TAA could be used to reduce aliasing and denoise at the same time.

8 References

Abi-Chahla, F. (2009) “Next-Gen 3D Rendering Technology: Voxel Ray Casting”. Available at: https://www.tomshardware.com/reviews/voxel-ray-casting,2423-2.html (Accessed: April 20, 2024).

Abrash, M. (2000) “Quake's Lighting Model: Surface Caching”. Available at: https://www.bluesnews.com/abrash/chap68.shtml (Accessed: July 12, 2024).

Acerola (2021) *I Made Minecraft Shaders*. Available at: https://youtu.be/dSS5HuSr4SQ (Accessed: July 16, 2024).

Akenine-Möller, T. *et al.* (2018) *Real-Time Rendering*. 4th ed. A K Peters/CRC Press.

Amanatides, J. and Woo, A. (1987) “A Fast Voxel Traversal Algorithm for Ray Tracing”. Available at: https://www.researchgate.net/publication/2611491_A_Fast_Voxel_Traversal_Algorithm_for_Ray_Tracing.

AMD (2022) “AMD FidelityFX™ Super Resolution 2”. Available at: https://gpuopen.com/fidelityfx-superresolution-2/.

Astle, D. and Hawkins, K.H. (2004) *Beginning OpenGL: Game Programming*.

Bikker, J. (2024a) “Ray Tracing with Voxels in C++ Series - Part 6”. Available at: https://jacco.ompf2.com/2024/05/29/ray-tracing-with-voxels-in-c-series-part-6/.

Bikker, J. (2024b) “voxpopuli.” *GitHub*. Available at: https://github.com/jbikker/voxpopuli (Accessed: June 4, 2024).

Bitterly, B. *et al.* (2020) “Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting”. Available at: https://doi.org/10.1145/3386569.3392481.

Boddy, Z. (2023) *Minecraft crosses 300 million copies sold as it prepares to celebrate its 15th anniversary*. Available at: https://www.windowscentral.com/gaming/minecraft/minecraft-crosses-300-million-copies-sold-as-it-prepares-to-celebrate-its-15th-anniversary (Accessed: March 27, 2024).

Careil, V., Billeter, M. and Eisemann, E. (2020) “Interactively Modifying Compressed Sparse Voxel Representations”. Available at: https://doi.org/10.1111/cgf.13916.

Cook, R.L. and Torrance, K.E. (1982) “A Reflectance Model for Computer Graphics”. Available at: https://doi.org/10.1145/357290.357293.

Crassin, C. *et al.* (2011) “Interactive Indirect Illumination Using Voxel Cone Tracing”. Available at: https://research.nvidia.com/publication/2011-09_interactive-indirect-illumination-using-voxel-cone-tracing.

D3D Team (2018) “Announcing Microsoft DirectX Raytracing!”, 19 March. Available at: https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/.

de Vries, J. (2014) *PBR Theory*, *Learn OpenGL*. Available at: https://learnopengl.com/PBR/Theory (Accessed: July 7, 2024).

Dohta, T., Osada, J. and Takayama, T. (2024) *Tunes of the Kingdom: Evolving Physics and Sounds for ‘The Legend of Zelda: Tears of the Kingdom’*. Available at: https://gdcvault.com/play/1034667.

Dwyer, D. (2022) *Drawing MILLIONS of voxels on an integrated GPU with parallax ray marching [Voxel Devlog #4]*. Available at: https://youtu.be/h81I8hR56vQ.

Dwyer, D. (2024) *Adding ray tracing (back) to my game engine [Voxel Devlog #17]*. Available at: https://youtu.be/aY4Zet_C9Zs.

Elwell, D. (2022) “DoonEngine.” *GitHub*. Available at: https://github.com/frozein/DoonEngine (Accessed: June 18, 2024).

Elwell, D. (2023) *Terra Toy. Steam*. Available at: https://store.steampowered.com/app/2435320/Terra_Toy/ (Accessed: June 18, 2024).

Elwell, D. (2024) *I'm Making a NEW VOXEL RAYTRACING ENGINE (in Vulkan) | Devlog 0*. Available at: https://youtu.be/7FNQ0QsoPOA.

EmbarkStudios (2020) “kajiya.” *GitHub*. Available at: https://github.com/EmbarkStudios/kajiya (Accessed: May 30, 2024).

ephtracy (2015) “voxel-model.” *GitHub*. Available at: https://github.com/ephtracy/voxel-model (Accessed: May 29, 2024).

fb39ca4 (2013) *Branchless Voxel Raycasting*. Available at: https://www.shadertoy.com/view/4dX3zl (Accessed: June 4, 2024).

Gore, E. (2024) *Demo Release and Q&A*. Available at: https://youtu.be/HsPb1765y2g.

Hjerpbakk, A. (2022) *Novel Extended Brickmap for Real-time Ray Tracing*. MA thesis, NTNU. Available at: https://hdl.handle.net/11250/3035458.

id Software (1992) “Wolfenstein 3D.”

Kajiya, J. T. (1986) “The Rendering Equation”. Available at: https://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf.

Khronos Group (2019) “Vulkan-Samples.” *GitHub*. Available at: https://github.com/KhronosGroup/Vulkan-Samples (Accessed: July 9, 2024).

Kilgariff, E. *et al.* (2018) “NVIDIA Turing Architecture In-Depth”, 14 September. Available at: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/.

Koch, D. *et al.* (2020) “Ray Tracing In Vulkan”, 15 December. Available at: https://khr.io/vkrayprovblog.

Laine, S., Marton, F. and Gobbetti, E. (2010) “Efficient Sparse Voxel Octrees”. Available at: https://research.nvidia.com/sites/default/files/pubs/2010-02_Efficient-Sparse-Voxel/laine2010i3d_paper.pdf.

Lars, K. (2023) *Palette Compression*. Available at: https://voxel.wiki/wiki/palette-compression/.

Lysenko, M. (2012) “Meshing in a Minecraft Game”. Available at: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ (Accessed: March 27, 2024).

Magnerfelt, C. (2012) “Transparency with Deferred Shading.” BSc thesis, EECS. Available at: https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group1Marten/final/final_TransparencyWithDeferredShading.pdf.

Maier, F. (2020) “dawn-ray-tracing.” *GitHub*. Available at: https://github.com/maierfelix/dawn-ray-tracing (Accessed: June 18, 2024).

Maier, F. (no date) *VoxelChain*. Available at: https://voxelchain.app/ (Accessed: June 18, 2024).

Mayes, K. (2020) *Vulkan Tutorial (Rust)*. Available at: https://kylemayes.github.io/vulkanalia (Accessed: January 4, 2024).

Mojang (2011) *Minecraft*. Available at: https://minecraft.net/.

Mulder, J.D., Groen, F.C. and van Wijk (1998) “Pixel masks for screen-door transparency”. Available at: https://doi.org/10.5555/869245.

Overvoorde, A. (2023) *Vulkan Tutorial*. Available at: https://vulkan-tutorial.com/ (Accessed: January 4, 2024).

Pharr, M., Jakob, W. and Humphreys, G. (2023) *Physically Based Rendering: From Theory To Implementation*. 4th ed. The MIT Press. Available at: https://pbr-book.org/.

Rundlett, G. (2022) *I made several visualizations using Vulkan in just 1 week, and here's how. (Daxa library showcase)*. Available at: https://youtu.be/b6uHmbWIMAQ.

Rundlett, G. and Dwyer, D. (2022) "gvox." *GitHub*. Available at: https://github.com/GabeRundlett/gvox (Accessed: May 29, 2024).

Ryan, J. (2022) "Teardown Teardown", February. Available at: https://juandiegomontoya.github.io/teardown_breakdown.html (Accessed: April 11, 2024).

Sakmary, M. (2023) *Atmosphere and Cloud Rendering in Real-time*. BSc thesis, CTU. Available at: https://github.com/MatejSakmary/atmosphere-bac-text.

Tuxedo Labs (2020) *Teardown Engine Technical Dive*. [Stream] Retrieved from YouTube.. Available at: https://youtu.be/tZP7vQKqrl8 (Accessed: April 11, 2024).

Tuxedo Labs (2022) *Teardown*. Available at: https://www.teardowngame.com/.

van Wingerden, T. (2015) "Real-time Ray tracing and Editing of Large Voxel Scenes."

Vandevenne, L. (2004) "Raycasting," *Lode's Computer Graphics Tutorial*. Available at: https://lodev.org/cgtutor/raycasting.html (Accessed: April 21, 2024).

vblanco20-1 (2020) *Vulkan Guide*. Available at: https://vkguide.dev/ (Accessed: February 22, 2024).

VG Insights (no date) "Teardown - Steam Stats". Available at: https://vginsights.com/game/1167630 (Accessed: June 28, 2024).

Villanueva, A.J., Karras, T. and Gobbetti, E. (2017) "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". Available at: http://jcgt.org/published/0006/02/01/.

von Funck, W. and von Funck, S. (2019) *Cube World*. Available at: https://picroma.com/.

Whitted, T. (1979) "An improved illumination model for shaded display". Available at: https://doi.org/10.1111/cgf.13916.

Wikipedia Contributors (2024d) *Octree*. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=Octree&oldid=1208044054 (Accessed: April 22, 2024).

Wikipedia Contributors (2024b) *OpenGL*. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=OpenGL&oldid=1218434754 (Accessed: April 12, 2024).

Wikipedia Contributors (2024a) *Path tracing*. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=Path_tracing&oldid=1195173925 (Accessed: March 21, 2024).

Wikipedia Contributors (2024c) *Vulkan*. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=Vulkan&oldid=1218520165 (Accessed: April 12, 2024).

Wittens, S. (2023) "Teardown Frame Teardown", 24 January. Available at: https://acko.net/blog/teardown-frame-teardown/ (Accessed: April 11, 2024).

Wolfe, A. (2020) *Casual Shadertoy Path Tracing 2: Image Improvement and Glossy Reflections*. Available at: https://blog.demofox.org/2020/06/06/casual-shadertoy-path-tracing-2-image-improvement-and-glossy-reflections/ (Accessed: July 4, 2024).

List of Tables

List of Listings

List of Figures

APPENDIX A: Interviews Notes

	Felix "Xima" Maier	Gabe Rundlett	Ethan Gore	Daniel "frozein" Elwell	Summary
What lead you to start working with voxels?	• side hustle • never liked triangle • sound more fun • like **minecraft** / pixel art • environment is not really dynamic • would love to see noita in 3D	• always interested • Minecraft ‣ Command blocks • Interactable world • Meshes don't make sense • Blender sculpting uses voxels	• Inspired by the original alpha of Cube World in mid-2021. • Enjoyed the feel of the voxel style.Started working with voxels in September 2021. • Continued due to the simplification of problems in voxel scenes, beneficial for a solo developer.	• other engine • opengl tutorials etc • videos	• Inspired by games like Minecraft and Cube World • Enjoy the dynamic nature of voxel worlds
For you, what is the hardest part when working with voxels?	• data structures • focus if there's a large or small world ‣ if theres a lot of dynamic stuff or not ‣ minecraft has a good scale with that • finding the right balance between rendering and updating data structure • people often expect GI • they still have a lot of potential • more used on the scientific side • compression slows down updating • plan: go full on simulation scale with a small world • Non sparse octree • octree are slow when looking for neighbors	• Too Much Data • Clever data structures ‣ Should be compact • Compressed on the fly within a brick • Tree on top of that • Fast random access • Very low memory footprint • Still researching • Questions about ray tracing: ‣ kajiya – Has world space irradiance cache – ReSTIR – Only one ray by 2x2 pixels for indirect – Shadow is only the sun with a shadow ray – Does spatiotemporal denoise	• Numerous voxel rendering techniques cause analysis paralysis. • Temptation to experiment with different techniques. • Data storage issues due to the volumetric nature of voxels.	• reduce memory consumption • naive ‣ store everything ‣ 2 level brickmaps ‣ was still too much • now more compact ‣ all data structures will be supported ‣ all chucks can have a different stucture	• Memory consumption • Tradeoff between memory and updating speed of data structure • Should have fast random access • Expectations of users (GI) • Number of rendering techniques can cause analysis paralysis

	Felix "Xima" Maier	**Gabe Rundlett**	**Ethan Gore**	**Daniel "frozein" Elwell**	**Summary**
	• water simulation will be cool • noita • 3d texture with mipmaps • texture are fast because they use texture swizzling • can be 30% faster than buffers	‣ Can be blotchy in dark areas – Low chance of a ray to hit the light – Solving this is hard – Would like the lighting for every voxel be uniform for every voxels – Light in voxel – Occlusion culling using raster ‣ Cellular automata has no noise – Uses a lot of memory			
From your personal experience, what do you think is the future of voxel rendering techniques ?	• hope they can make the world more dynamic and have bigger world • more realistic minecraft • Brickmaps ‣ Better balance	• Not enough experience • Towards ray tracing • Xima CO can use light for gameplay elements, not with ray tracing • Would be interesting to see if raytracing could have cellular automata qualities for gameplay	• Efficient primary ray rendering through standard rasterisation with proper culling. • Potential combination of rasterisation for primary rays and ray tracing for secondary rays. • Dual acceleration structures increase memory and generation time but may reduce ray usage.	• Still a lot of room • john lin • not sure • still exploring	• Hope to make worlds more dynamic • Inspired by John Lin • Brickmaps have a good balance of memory / update speed • Light computation technique could impact gameplay (e.g. cellular automata) • Use rasterisation for primary rays and ray tracing for secondaries
Based on what you've already done, what would be your suggestions to improve current voxel rendering?	• We are at a limit in terms of compression • John lin was using a compression technique that was invented 10 years ago • Regarding denoisers and TAA they could profit from	• Storage could be improved • a lot of research for medical and movies (clouds) • Most popular by far is OpenVDB ‣ Similar to an Octree but with 8x8x8 or 16x16x16 with bitmasks	• Do not overlook rasterisation; it's effective. • Ray tracing offers more interesting problem-solving opportunities.	• GI ‣ old engine wasn't great ‣ not started on new • speedup thanks to voxels	• Don't agree on whether storage could be improved or not (possible they don't talk about the same metric) • Denoisers could be designed around the simple shape of cubes

	Felix “Xima” Maier	Gabe Rundlett	Ethan Gore	Daniel “frozein” Elwell	Summary
	the fact that cubes are simple shapes • Xima has a denoiser like that, but more could be done • Voxels are easy if you need to do filtering • Make lighting constant across faces • Gives a pixelated lighting • 20 lines of code • Upscaler ‣ Uses a checkerboard upscaler ‣ use the world space and normals • With voxels it’s easier	‣ Would be the best place to start ‣ DAGs for static volume data ‣ Nanite vertex data is turned to a DAG ‣ Delta compression, RLE • If you want a tree with swaying leaves ‣ Could bake it ‣ Could have a bone system that could be modified and could be voxelized with SDFs ‣ Could be too slow for an entire forest			• Upscaling could also profit from cubes • Rasterisation should still be considered
Have you used HWRT? If yes, what is your personal experience with it related to voxels?	• 5 years ago, added HWRT to WGPU • Too slow for individual voxels • Chunk based might be good • HWRT is overkill • the TLAS is not optimal for AABB • it would be faster to write yourself • not limited to modern hardware • HWRT is a huge blackbox • John lin ranted about it ‣ He gave up HWRT • Ray sorting is is not very usefull with chunk based HWRT	• Still learning • You can just pass a sparse representation • It only makes sense to give data from the surface • It’s the same problem as meshing • Updating the TLAS is anoying because it adds cost • You can have anything in the “BLAS” with a custom intersection test • For trees you could evalutate the SDF the tree • Didn’t try to have multiple Data structure in the BLAS • Only has a 8x8x8 brick using bitmasks	No experience.	• HWRT will help • Is faster for now on big maps ‣ small and big volumes • was sceptical • now thinks will be faster • Next step will be shadows • particles and waters are rasterised	• No consensus on if HWRT can be faster for voxels • The imposed “black box” TLAS is not designed for voxel data, one written by hand can be faster • Passing only surface data helps • Can be faster on big maps • HWRT is designed for the divergent nature of ray tracing (avoid threads being idle like in compute) • Ray reordering helps even more • Work graphs could help with that, but BVH traversal is slower with them

	Felix “Xima” Maier	**Gabe Rundlett**	**Ethan Gore**	**Daniel “frozein” Elwell**	**Summary**

	• Mixing two hardware units could be a problem
	• BVH stuff is hard and boring
		• Prefers to do everything in compute
		• But HWRT is designed for the divergent nature of ray tracing
		• In compute, you can have threads finish early and have the rest work
		• Makes it so that the GPU has a lot of doing nothing
		• HWRT has a queue of work items and gives these works to utilise the GPU better
		• Makes it so you have better saturation of the GPU
		• New NVIDIA GPUs have shader excecution reordering
		  ▸ You can give a key value to the GPU and the GPU will resort that key to be in warps
		  ▸ They have the same cacheline
		  ▸ Could hit the same side of the branch (exit of the loop)
		  ▸ Can improve by up to 2x
		• NVIDIA cannot overlap any work on a single queue
		  ▸ Daxa doesn’t have multi queue
		• Work graphs (only DX12 and AMD Vulkan) Could help for terrain generation			

	Felix “Xima” Maier	Gabe Rundlett	Ethan Gore	Daniel “frozein” Elwell	Summary
		▸ BVH traversal with workgraphs is slower than HWRT • Ray tracing work through trace could be use for example for terrain generation but haven’t tried yet			
If you had to restart your project from scratch, is there anything that you would change?	• Not really • It used to be a cpu-based voxel engine • Since he targets large scale world simulation ▸ GPU Driven design ▸ Would have been nice to start from that ▸ The shift was compilcated, but now it’s good • Has a fork where it’s easy to prototype stuff ▸ If things a good, it is merged to the main branch • GPU driven is hard to protype in (gabe rundlett gave up) • HWRT can start shaders from shaders ▸ New extension might be better since it’s made for that	• New to software architecture design • Still learning • Wouldn’t start again • Keep learning and improve it in a continuous maner • For ex: terrain gen should be independant from rendering • Should be able to rip out part of the project	• Reconsider the scale of the engine. • Smaller scale might allow for testing new ideas not feasible at current scale.	• different GI • two compute shaders ▸ primary, gets colours ▸ then lighting shader for every visible shader ▸ each voxel was 16 bytes	• GPU Driven rendering lends itself well to voxels • Having a second branch for rapid prototyping can help • Think of how big the project should be early on • Try to keep big modules separate (eg, terrain gen should be able to be changed without breaking rendering)
What is your personal experience with voxel animations ?	• Models made and animated with blockbench https://www.blockbench.net/ • Models are voxelized in a 3d texture	• Unexplored territory • Looked into animating pixel art ▸ lot of things to learn from that • Still a lot to learn	No experience yet.	• was one of the hardest part • required to update the data continuously • the hardest part was allowing people to update stuff	• This is a hard problem • Can voxelize mesh model ▸ Render entities to quads on screen • Can use a distance field

	Felix "Xima" Maier	**Gabe Rundlett**	**Ethan Gore**	**Daniel "frozein" Elwell**	**Summary**
	• Each entity would need its own 3d volume • 200k no problem to render • models in blockbench format with code based animations • jem -> voxel • ~11 Compute passes • Entities was harder than raytracing • Rendered using quads on the screen • entities don't have lighting yet			• stored a DF of each objects to check what objects people clicked on • keyframe ‣ one volume per anims ‣ have to save in flat array • maybe voxelize meshes ‣ depends on size • static: dag • dynamic: brickmap or svo	
Is there anything we didn't mention on the subject of voxel rendering that you think would be worth mentioning ?	• GPU driven rendering design is well suited for voxels • Large scale simulation friendly • Cellular automata is helped by the gpu driven rendering	• no • Most important is "Everything is dynamic" • If you're making a voxel project you should focus on the interactivity of the voxels		• impressed by the amount of work done in the space • have a lot of potential • not enough research	• Focus on dynamic worlds