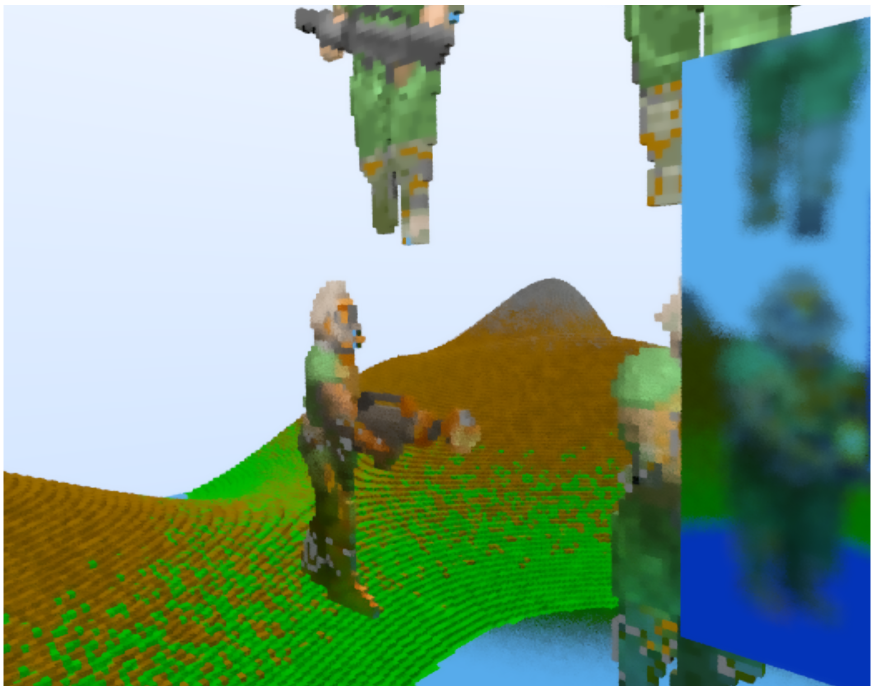
Aksel Hjerpbakk

Novel Extended Brickmap for Real-time Ray Tracing

Master's thesis in Master of Science in Informatics
Supervisor: Anne C. Elster
June 2022



Aksel Hjerpbakk

Novel Extended Brickmap for Real-time Ray Tracing

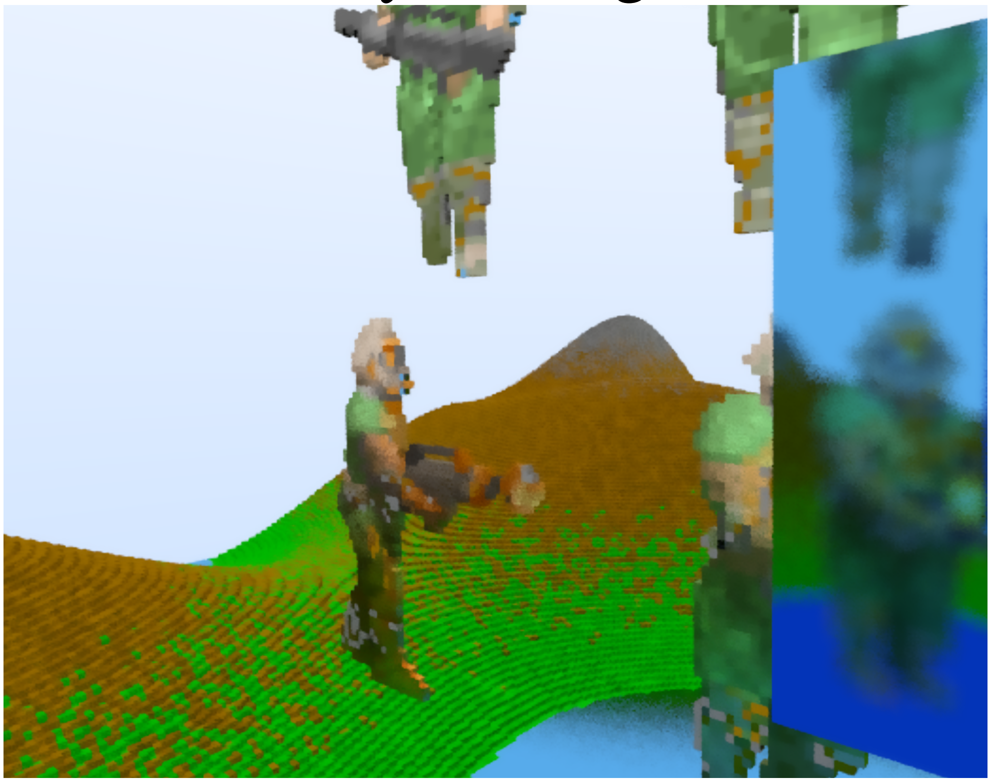
Master's thesis in Master of Science in Informatics
Supervisor: Anne C. Elster
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Aksel Hjerpbakk

Novel Extended Brickmap for Real-time Ray Tracing



Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
CC-BY June 13, 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

HPC
LAB

Acknowledgement

I would like to express my deepest gratitude to Anne C. Elster and the HPC-Lab at NTNU for sharing their knowledge and resources with me. I am also very grateful to other researchers who share their knowledge with the public, especially Peter Shirley, who took the time to answer any question I sent to him and wrote the fantastic *Ray Tracing in One Weekend* book series.

I would also like to mention the open-source community. I would not be able to complete this thesis without projects like Zig, mach-glfw, and vulkan-zig.

Finally, a sincere thanks should also go to my loving partner, family, and friends for their moral support throughout this process.

Abstract

Storage and rendering of large-scale volumetric data can be a challenge. In this thesis, we will expand on the features of a data structure called *brickmap* from 2015. The brickmap data structure is a solution that delivers a practical method for storing large quantities of voxel data. It is suitable for manipulation and ray tracing in real-time. This thesis will explore how the structure can be implemented in Khronos's Vulkan compute API and extend to include physical-based material data while remaining practical in real-time applications for higher fidelity ray tracing and path tracing.

A benchmark system was created to test the performance of the brickmap implementation in a controlled environment across different hardware. The results generated from the benchmark system show that the previous generation's graphical processing units can ray trace in real-time when using the brickmap data structure if the correct optimizations are applied. The weakest card tested, the GTX 1650M, showed promise but failed to scale with the current solution, while other cards like the GTX 1080ti were able to perform in all our tests. Newer cards like the RX 6800 XT were not stressed in the benchmark and were bottlenecked by CPU overhead. The thesis also includes several ideas for future work.

Sammendrag

Lagring og tegning av storskala volumetrisk data kan være en utfordring. I denne avhandlingen kommer vi til å videreutvikle en datastruktur kalt *brickmap* i fra 2015. Brickmapdatastrukturen leverer en praktisk metode for å lagre store mengder med voxel data på. Datastrukturen har også god støtte for å strålespores og manipuleres i sanntid. Vi skal se på hvordan denne strukturen kan bli implementert i Khronos sin grafikk API, og videreutvikle strukturen til å støtte fysisk basert materialdata. Den videreutviklede brickmapstrukturen skal også forbli praktisk for manipulering og strålesporing i sanntid.

Det ble bygd et system for å teste implementasjonen av brickmapstrukturen. Dette systemet gjorde det mulig å teste ytelse på tvers av maskinvare i et isolert miljø. Våre funn viser at GPUer I fra tidligere generasjoner er i stand til å tegne med strålesporing i sanntid gitt at de rette optimaliseringene er brukt. Det svakeste kortet vi testet var GTX 1650M. GTX 1650M viste lovende resultat, men den var ikke i stand til å skalere. GTX 1080 Ti ble også testet og viste gode resultat selv med mer stressende arbeidskrav. Mer moderne kort slik som RX 6800 XT ble ikke utfordret i testene som ble utført og var begrenset av CPU. Flere ideer for fremtidig arbeid vil også være inkludert.

Contents

Figures

Tables

Code Listings

Acronyms

ALU Arithmetic-Logic Unit. 26

API Application Programming Interface. xix, xx, 3, 8–11, 13, 18, 22–26, 39

BSDF Bidirectional Scattering Distribution Function. 13

CI Continuous Integration. 42

CLI Command Line Interface. 25

CPU Central Processing Unit. 2, 8–11, 14, 18, 19, 21, 24, 26, 27, 34, 39

DDA Digital Differential Analyzer. 2, 8

FFI Foreign Function Interface. 22, 25

FPS Frames Per Second. 36

GLSL The OpenGL Shading Language. xix

GPGPU General-Purpose Graphics Processing Unit. xix, xx, 3, 8, 10, 11, 23

GPU Graphics Processing Unit. xix, xx, 2, 3, 6, 8, 9, 12–15, 18, 19, 23–27, 33–37, 39, 40

GUI Graphical User Interface. 21, 22, 24, 25, 31

HDR High Dynamic Range. 41

IDE Integrated Development Environment. 29

ISA Instruction Set Architecture. 26

LOD Level Of Detail. 7, 8, 15, 18

MIMD Multiple Instruction Multiple Data. 9

MISD Multiple Instruction Single Data. 9

OS Operating System. 13, 24

PBR Physical Based Rendering. 2

RAM Random-Access Memory. 8, 9

RGA Radeon GPU Analyzer. 25, 26

RGP Radeon GPU Profiler. 25

RMA Radeon Memory Analyzer. 25

SDK Software Development Kit. 40

SIMD Single Instruction Multiple Data. 9

SIMT Single Instruction Multiple Thread. 9

SISD Single Instruction Single Data. 9

SPIR Standard Portable Intermediate Representation. xix

SVO Sparse Voxel Octree. 2, 6

VRAM Video Random-Access Memory. 9, 37

WIP Work In Progress. 22

Glossary

Cuda A proprietary GPU API which enables General-Purpose Graphics Processing Unit (GPGPU) compute for Nvidia hardware. Made by Nvidia. 10, 11

culling When objects are removed from the draw list based on different factors in order to perform faster shading. Frustrum culling removes objects based on if is visible to the camera. Occlusion culling will discard objects that are blocked by objects in front of them and therefore not visible.. 6, 8

DirectX A proprietary GPU API which enables rendering and GPGPU compute on windows machines. Made by Microsoft Corporation. 9, 23

GLSL The OpenGL Shading Language (GLSL) is a shading language used by some graphics APIs to define fragment, vertex, compute -shading and more. The language has a C-like syntax. 15, 33

Metal A proprietary GPU API which enables rendering and GPGPU compute on macOS machines. Made by Apple Corporation. 23

OpenCL A open source GPU API which enables GPGPU compute for different hardware. Made by Khronos Group and contributors. 10

OpenGL A open source GPU API which enables rendering and GPGPU compute. Made by Khronos Group and contributors. 7, 9–11, 23

OpenMP A open source multi processing API. Made by OpenMP Architecture Review Board. 11

rasterization A technique used to project 3D geometry onto a 2D plane and subsequently draw it to a screen. 1, 2, 13, 41

ray tracing A technique used to draw 3D geometry onto a 2D plane by tracing rays through a scene and testing intersection of said rays with the given 3D scene. 1–3, 5–8, 10–14, 23, 31, 33, 37, 39–42

SPIRV Standard Portable Intermediate Representation (SPIR) is the target shader language for vulkan. It's a intermediate language which means that is normally not hand written, but a compile target for other languages. 15

Vulkan A open source GPU API which enables rendering and GPGPU compute designed for modern hardware and easier vendor support. Made by Khronos Group and contributors. 3, 10–12, 15, 19, 22, 23, 25–28, 39, 40

Chapter 1

Introduction[1]

The work presented in this paper requires some fundamental background knowledge about the rendering field. This chapter introduces some fundamental rendering concepts. The concept of rasterization, polygons, ray tracing, voxels, and data structures for voxels will be covered. This chapter will also present our contribution and an outline of the thesis.

Rendering is a broad field with many different techniques. The traditional rendering pipeline utilizes rasterization to project 3d data onto the screen convincingly. This method has the advantage of being relatively fast compared to other methods. A common disadvantage of rasterization is that it is harder to achieve visual effects that depend on non-local attributes relative to the screen. Specific examples of such challenges are global/indirect illumination and reflection. For most modern rendering, we use polygon-based models. Polygon models are usually triangle-based. The triangles connect in order to create a mesh. It is relatively fast to render polygons using rasterization, but it can be challenging if the mesh's topology has to be dynamic.

Ray tracing is an alternative to rasterization where a scene is drawn by tracing rays that travel through it and testing what each ray hits to give an accurate color representation. This method makes it easier to simulate real-life concepts like reflection and shadows by fundamentally being closer to how natural light behaves. The downside of ray tracing is that the render speed is usually magnitudes slower than traditional rasterization. Ray tracing tend to be slower because it requires simulating a large number of rays in order to get a clear image. Hardware and software for rasterization have also been more developed than ray tracing, especially for real-time rendering. Usually, memory access and intersection testing are the biggest culprits of render time for ray tracing. Rendering a single frame usually requires millions or billions of ray intersection tests using ray tracing. In order to reduce intersection testing to a minimum, the scene has to exist in an accelerated data structure to avoid wasteful hit tests.

Polygon-based meshes only describe surfaces. Voxels are an alternative to poly-

[1]This thesis includes content from the specialization project IT3915 & TDT24 for Aksel Hjerpbakk, both submitted in autumn 2021.

gons and can describe volumes. Volumes are beneficial in the medical field, fluid simulations, and more. *Minecraft* is a famous voxel-based video game that allows players to manipulate the voxel world in real-time. Voxels are *3D pixels*. Much like each pixel in a 2d image represents color, a voxel can represent color (or lack of) in 3D. Many voxel data structures are beneficial for ray tracing. These data structures enable accelerated ray tracing traversal. A regular 3D grid can represent a set of voxels, but this has the downside of using large amounts of memory as vast empty spaces still take as much memory as occupied entries. Even with fast grid traversal algorithms like Digital Differential Analyzer (DDA) the number of lookups for a ray might cause issues in wast empty areas when using a uniform voxel grid.

Sparse Voxel Octree (SVO) is a tree data structure that can represent voxel grids. A 3D SVO consists of a root node with either 0 or 8 children, and each child node has either 0 or 8 children. The max depth of the tree is predetermined and dictates the granularity of each leaf node/voxel. This data structure enables rays to travel through significant gaps in space and reduces the amount of wasteful memory access compared to simple 2D arrays. There is some downside to SVOs. It takes more time to manipulate the leaf nodes of a SVO than a simple 3D grid. They can also be problematic when rays travel parallel to a defined voxel surface over large distances as they have to perform a significant amount of granular lookups. This scenario is more computationally expensive than a simple 3D grid. Granular traversal is mitigated by adding "pointers" to each neighboring node which has the downside of increasing memory usage.

The original brickmap thesis describes an alternative data structure that attempts to accelerate ray tracing traversal while supporting efficient large-scale editing of the grid data[1]. Brickmaps have the added benefit of enabling data streaming between CPU and GPU. The storage scheme of the brickmap algorithm also reduces the data size of individual voxels. Each voxel is an entry in a *brickmap*. On top of the brickmaps is the brickgrid, which stores brickmaps. A brickmap contains a bitmask. The bitmask is 512bit wide, and each bit of the value describes if a voxel exists at a given location. One voxel in the brickmap is approximately 1 bit in size. Brickmaps are assigned a start index and can point to color data in a global color array. The brickmap start index is added with the bit offset of the voxel to find the color of a given voxel. The brickmap thesis uses DTX1 compression for color storage to reduce the added memory footprint of color storage.

Material information is usually used for ray tracing as it enables surfaces to have properties that lead to more realistic ray-to-surface interaction. It is also used in traditional rasterization renderers that utilize Physical Based Rendering (PBR) methods for more realistic light simulation. This information may include surface attributes such as type (metal, diffuse, dielectric) and surface roughness. Suppose voxels are to include unique material information. A lossy compression scheme cannot be utilized like the original brickmap paper since attributes such as material type must be truthful. When a voxel is assigned a glass material type, it would be jarring to observe lossy compression, causing the material type to change to metal or something other than the glass type. The original brickmap paper presents a reliable and effective method for storing voxel geometry but leaves much potential for improved color and material storage work. We present a new method for storing color and material information while maintaining its usefulness in real-time applications.

1.1 Contribution

This paper describes the implementation of a real-time ray tracing application utilizing software-based General-Purpose Graphics Processing Units (GPGPUs) techniques. By not using ray tracing hardware acceleration, the application can run on older generation GPUs. We also present a material and color storage scheme that extends the capabilities of the original brickmap data structure described by Thijs van Wingerden[1]. Finally, the thesis presents a proof of concept implementation. The implementation uses the Vulkan Application Programming Interface (API) and currently runs on Windows and most Linux distributions. This thesis also documents relevant implementation challenges and solutions. The practical aspects of developing the application are covered to help developers and researchers that share the goals of GPGPU ray tracing.

1.2 Report outline

In Chapter 2.1 we cover some background information and discuss related work relevant to ray tracing. The thesis also includes background and related work for graphic APIs and design of ray tracing rendering in Chapter 2.2. Chapter 3 presents a new material and color storage scheme that extends the brickmap data structure. The chapter will also present our brickmap implementation with the mentioned storage scheme. Chapter 4 discuss the implications of different ray settings. 4 also presents our application benchmarking result and discusses the findings. The last section is the conclusion and future work Section 5.

Chapter 2

Background & Related Work

This chapter discusses the background and related work on which this thesis is built. First, we describe ray tracing, the main application of our work, and the data structures associated with ray tracing, followed by fast voxel-ray intersections and the original brickmap algorithm. Section 2.2 describes the render system, including an overview of CPU and GPU hardware and benchmarks for ray tracing on GPU APIs, Vulkan benchmarks, and particle simulations.

Some of the content in this chapter were part of the author's hand-in in TDT24

2.1 Ray Tracing

Ray tracing is a simple concept. On a fundamental level, backward ray tracing fire rays from the camera or eye perspective and test what the rays hit. The rays can bounce at their hit points for more realistic image output. However, this concept can be very complicated, especially when real-time performance is required. Some of the issues related to ray tracing are not obvious when initially learning about it. This chapter will give some background knowledge about accelerated data structures, an essential concept for accelerated ray tracing. The chapter also looks into research for a fast ray-to-slab intersection method. This slab intersection method can accelerate the ray tracing of voxels.

2.1.1 Accelerate ray traversal with data structures

One of the challenges with ray tracing is scene traversal and performing intersection hit tests that are not wasteful. For example, the method used by the research discussed in Section 2.2.2 is to increment a ray position by a small amount and test if the point is inside an object. There are two problems with this method:

- How far the ray should travel is unknown; therefore, small steps have to be taken for each increment, causing many increments for considerable distances. Repeated increments will cause substantial rendering overhead.

- Each ray increment has to perform intersection testing on all global or local objects. What objects might intersect with the ray is unknown, so it has to test against all objects in the scene unless a hit is detected.

An optimal solution should be able to describe a ray as a function and test if the combination of direction and origin correlates with an intersection instead. Using this principle, we no longer need to perform increments of a point along a ray or reduce the number of increments. It should also be able to perform culling on as many objects as possible before intersection testing. Some sort of culling is vital to be able to populate a scene with more than a few objects.

Sparse Voxel Octree (SVO) is a data structure that utilizes octrees to calculate ray increment size. An SVO also has information about what objects are possibly intersecting with a given ray, much like a traditional scene graph can be used for culling rasterization. Ray tracing is a parallelizable problem. To achieve real-time ray tracing using a SVO it should utilize the GPU. Tree structures can be challenging to use on a GPU because they usually rely on pointers or references. A GPU friendly SVO is presented in the book "GPU Gems 2" called a $N^3 tree$[2]. The suggested use case was texture mapping combined with volumetric simulations. A demo application showcase these use cases. $N^3 tree$ also includes an optimized lookup algorithm and effective data storage using sequential memory, which is a requirement on the GPU.

The demo application for the $N^3 tree$ allows users to paint on a 3D mesh using different colors. The applied color will then be simulated as a surface liquid and flow downwards along the mesh in real-time. The application was written between 2003 and 2004 and shows that real-time changes to the octree are possible.

Although SVO has proven to be a viable data structure, there is an overhead to traversing and manipulating the tree in real-time. The brickmap data structure is an alternative to the SVO data structure with good ray tracing acceleration performance while remaining viable for grid manipulation in real-time. We discuss the brickmap data structure in Section 2.1.3.

2.1.2 Fast Voxel-Ray Intersection

Voxels are highly relevant when discussing ray tracing as it is a volumetric format that often fits with ray tracing algorithms. By assigning local triangles to a voxel, voxel data structures can accelerate ray traversal when rendering traditional polygon-based models. Voxels also have many benefits over traditional polygons because of their volumetric nature and can be used for scientific and recreational applications.

Voxel ray tracing applications exist, either as vaporware or released products. Most of them are proprietary without open-source code. An example of a proprietary application is *Teardown*, which is a physics-based voxel game that utilizes ray tracing and does not require dedicated hardware[3]. The closed nature of these projects means that even though it displays the possibilities of voxel ray tracing, it does not add to the knowledge pool.

Majercik et al. proposed an efficient voxel ray tracing technique in 2018. The new technique does not utilize specialized hardware, even though Nvidia researchers made the paper. It presents a new slab intersection method for fast intersection testing that calculates the intersection normal vector and distance. The slabs are also not required to be axis-aligned. It also presents two implementations of voxel renderers that utilize the mentioned intersection test. Finally, they present actual benchmarks of the applications to prove the intersection's performance.

The slab intersection method was slower than one previous method, but the slab intersection was always faster when including normal vector and distance calculations. Normal vector and distance data are essential when performing more advanced shading like volumetric rendering of fog and smoke. Normals are also required when performing ray tracing bounces. It was implemented in Unreal to display the possible applications in games. The authors also implemented a OpenGL solution to show the viability in non-game applications. The OpenGL solution utilizes a specialized rendering pipeline.

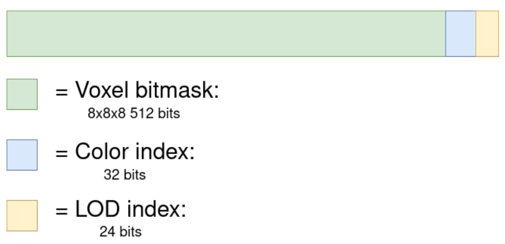
The implementation can be reminiscent of the True Impostor algorithm presented in the book GPU Gems 3[4]. `GL_POINT` is used to represent individual voxels. The vertex stage calculates a bounding quad like in the Impostor algorithm, and the fragment stage utilizes this quad to calculate the 3D cube represented by the point using ray tracing. The implementation is unique in how it solves voxel rendering. The rendering implementation also proves to be faster than the reference rasterization renderer. There are limitations to the implementation. The render does not trace rays. The screen emits rays that test if they hit any voxel. Any further ray bounce simulation could present technical issues since voxel quads generate relative to the camera. The lighting techniques they use are traditional rasterization methods with shadow maps. However, the intersection method presented is versatile and can be used to implement a pure ray tracing renderer.

2.1.3 The original brickmap algorithm

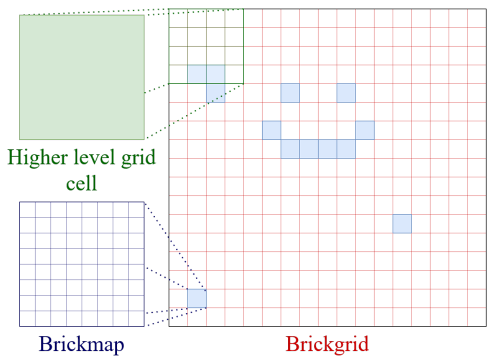
The brickmap voxel data representation is a hierarchy of uniform grids. The grids contain raw voxel data at the lowest layer, called brickmaps. Brickmaps encode solid data in a bitmask and store color separately[1].

It is important to emphasize that the voxel's memory footprint must be as small as possible. We want to store as many voxels as we can in memory. Less memory usage means potentially more voxels and reduces the probability of cache evictions as more entries can exist in local memory. Figure 2.1a shows how the mentioned bitmask is tightly packed with a color start index and a Level Of Detail (LOD) index if unique voxel color information is not needed. The brickmap start index plus the bit offset of the voxel results in the voxel's color index.

Brickgrids store brickmaps. The brickgrid is illustrated in figure 2.1b. The brickgrid does not store the brickmaps but indices to brickmaps. The index storage requires less memory than actual brickmap storage while traversing. The original



(a) Brickmap memory layout



(b) 2D brickgrid data structure. Cells that have a blue fill color illustrate defined brickmaps while empty cells are not yet assigned any values

paper uses DDA to traverse the grid. DDA is an algorithm to interpolate over an interval defined by a start and end point but can be modified to work with a start point and a direction which is helpful for ray tracing.

A higher-level grid accelerates ray traversal over a sizeable empty distance. If one or more of the brickmaps in the cell is not empty, then the cell is set. Data streaming between the CPU and GPU and vice versa in the structure is an expected use case. This is useful for LOD, culling, chunk streaming, and so on.

2.2 Render System

Achieving optimal performance depends on a multitude of different factors. One of these factors is utilizing hardware accordingly. Efficiently utilizing hardware requires background knowledge. This chapter will describe the core components of a modern computer, cache utilization, execution models, and more. We will also discuss research related to graphics APIs and how different APIs and API features might have performance implications.

2.2.1 GPU, CPU & other hardware

A modern computer consist of many components, but the main ones are: Graphics Processing Unit (GPU), Central Processing Unit (CPU), motherboard and Random-Access Memory (RAM). The GPUs role is to draw to the screen, but modern ones have functionality for generalized computing. General-Purpose Graphics Processing Unit (GPGPU) is suitable for problems that consist of many smaller problems that are similar and therefore parallelizable. The CPU is the main processing unit for a modern computer. The CPU solves problems that are not as parallelizable and therefore only need one or a few threads compared to the thousands of threads on a modern GPU. A CPU can also do sequential work magnitudes of time faster than a GPU. For example, an AMD Ryzen 9 5950x CPUs has a max frequency of 4.9GHz while an AMD Radeon RX 6900 XT has a boost speed of 2.3GHz.

Another major difference between a CPU and GPU is their execution model. When talking about execution models we use Flynn's taxonomy to describe these models. Flynn's taxonomy has been extended, but the the initial definition included 4 classifications[5]:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

A modern CPU uses the MIMD and SIMD execution model. GPUs uses the Single Instruction Multiple Thread (SIMT) execution model which is part of an extended taxonomy. Using Nvidia's terminology, the GPU consist of *grids* of *blocks*. The *blocks* has *warps* and the *warps* has *threads*. This is true for most GPUs including non Nvidia models although the terminology differs.

Warp divergence can emerge on the GPU because it uses the SIMT model. Warp divergence occurs when threads branch into different parts of the code. The divergence forces the warp to execute both code branches in sequence and effectively adds both execution times to the total run time.

The CPU and the GPU tend to have cache memory. This memory is close to the processing unit of the component and is cheap to read. Random-Access Memory (RAM) can also store memory. The RAM is usually much larger than any component cache. However, modern GPUs usually has similar amounts of memory, called Video Random-Access Memory (VRAM). Because of this, RAM and VRAM are usually the primary storage of runtime memory depending on the component and is subsequently cached on the components when needed. Data is transferred through the bus on the motherboard when a program access RAM data. This transfer is a slow operation and usually will take many clock cycles to complete. AMD describe this phenomenon on the GPU in one of their articles:

> *A GPU like Radeon™ Fury X (...) has 512 GB/s of [local] bandwidth (...) but the PCIe (3.0) bus supports at most (...) a sum of 32 GB/s in both directions*[6]

From this quote, we can conclude that 1 to 10 clock cycle instructions are generally significantly cheaper than any memory retrieval that can be thousands of clock cycles. It is essential that the memory usage is kept at a minimum and has a sequential access pattern to increase the chance that memory usage results in a cache hit instead of evictions and subsequent retrieval.

2.2.2 Benchmarks for Ray Tracing on GPU APIs

A modern GPU is very flexible in performing algorithms and visualizing the results. What API and concepts the programmers should choose to utilize is not necessarily obvious. When considering graphics APIs like OpenGL or DirectX, there are multiple entry points for ray tracing since the fragment stage, and compute pipeline/stage are both viable.

Benchmarking data on the fastest APIs and methods in the context of ray casting volumetric scenes is presented by F. Sans et al. [7]. The method used is a simple ray casting with different APIs and concepts to find the most performant options. The paper present a implementation for each target to benchmark, these targets include: OpenGL fragment shader, OpenGL compute shader, Cuda and OpenCL. They used three volumetric data sets to test the APIs. Since the research focuses on benchmarking ray casting in the context of APIs and methods, their implementation is a ”brute force” based ray traversal where each ray traverses with a small predefined increment distance. This implementation has problems with more complex scenes. The benchmark result shows that bigger scenes scale poorly, and performance degrades quickly with scene size in all cases.

Their findings show that OpenGL compute shaders is the most performant option compared with Cuda, OpenCL and OpenGL fragment shader[7]. Since our research is focusing on making real-time ray tracing with Vulkan, this can indicate Vulkan compute pipelines might be a viable option compared to utilizing the graphics pipeline fragment stage. A compute pipeline is also more flexible than a fragment stage. Compute pipelines can solve various problems using arbitrary inputs and outputs. A fragment stage in a graphics pipeline is bound to produce a fragment and has stricter requirements on input data.

2.2.3 Vulkan API Benchmarks

PolyBench is an open-source benchmarking suite to test against compilers, APIs and environments. It enables objective performance comparison between them. The benchmark focuses on problems related to linear algebra and more. Linear algebra happens to be one of the main components when programming ray tracing software.

PolyBench/GPU is a PolyBench implementation using GPGPU APIs where version 1.0 includes Cuda and OpenCL. There is also a Vulkan implementation that is called vkpolybench presented by N. Capodieci et al[8].

vkpolybench offers a compute abstraction that enables others to extend the benchmarks if needed with new algorithms without having to deal with some of the Vulkan's repetitive and verbose aspects. The same authors created the compute abstraction in a previous work[9]. The abstraction aims to create an easy-to-use CPU-to-GPU dispatch API that in many ways mimics Cuda while still preserving the control that Vulkan offers. It is worth noting that most abstraction will come at the cost of control and, in many cases, performance, but the findings included in their research show the framework beating Cuda in all benchmarks. The authors have the following statement about the results:

> *Our findings show that no matter which submission model is selected for Cuda, Vulkan bare metal approach manages to minimize and better distribute the CPU overhead*[9]

vkpolybench makes it easier to compare the API layer overhead with Cuda, OpenGL and even performance of CPU APIs like OpenMP. Much like the CPU-to-GPU dispatch research, the results presented by vkpolybench are promising for Vulkan's performance. According to vkpolybench, Vulkan-compute performs on average better than alternative APIs. We can conclude that implementation of ray tracing using compute can utilize Vulkan for optimal performance in the API layer based on the results presented in the code repository readme[8].

2.2.4 Particle simulations using asynchronous compute

K. Enarsson presents potential gains from using asynchronous compute for particle simulations[10]. Asynchronous execution is one of the fundamental features that Vulkan presents in its execution model[11]. Because of this, looking at how asynchronous execution affects performance in other APIs can help give an idea of the potential gains of asynchronous execution in Vulkan and if taking advantage of such a feature is worth the added complexity. The paper focuses on benchmarking asynchronous DirectX 12, but changes in performance between asynchronous and synchronous DirectX 12 calls should be representative for Vulkan as well. They utilize a GPU compute N-body physics kernel and benchmark how executing the kernel synchronously compared with asynchronously affects performance. In order to avoid overfitting, the author uses 3 GPUs, AMD RX560, Nvidia GTX 1060, and one GPU that does not officially support asynchronous compute execution: Nvidia GTX 960.

The scheduling of asynchronous workload works by populating idle blocks that would otherwise remain unused between work dispatches. Problem sets with uneven workloads for each block would probably gain performance with asynchronous dispatch. The potential gains diminish when the workload increases since the GPU compute blocks will have diminishing idle time for a given dispatch. The paper finds that both the 560 and 1060 have considerable performance gain when the problem set is minor, with over a 30% increase in performance on the smaller problem set. Interestingly the 960, which does not officially support asynchronous work, does see performance improvements at around 11% on the most trivial workloads. There is also the aspect of memory retrieval to consider. Bigger workloads tend to work with more significant memory requirements, and therefore, memory retrieval might cause slower computation as warps start fighting for memory.

The value of asynchronous compute for ray tracing

In our opinion, there are two main aspects to the research findings[10] to consider in the context of GPGPU based ray tracing:

1. Ray tracing is very expensive. Performance gains will be diminishing accordingly[10]. However, the findings indicate small gains even on big workloads.

2. With multiple bounces, ray tracing starts to become bottlenecked by memory access and computations. The bounces are random, making each warp dependent on a broader range of buffer memory as rays start dispersing and require different parts of the scene state.

The next generation of hardware is faster than the previous generations, as observed in their result when comparing the GTX 1060 with the 960. These improvements might suggest that newer cards will improve performance with asynchronous execution. Not only does GPU tend to improve, but also the drivers that power the cards.

Based on these findings, we will also investigate how using asynchronous computing could be used for ray tracing. However, as described in Section 4.3.1, we show that one can lose performance using asynchronous compute. The following chapters thus focus on the other techniques we developed for real-time ray tracing, including improved memory utilization in Vulkan and the new material storage scheme for the brickmap data structure.

Chapter 3

Real-Time Rendering through Ray Tracing

This thesis aims to achieve real-time performance for ray tracing without hardware acceleration but by taking advantage of modern GPUs generalized compute capabilities. This goal is not trivial since ray tracing is computationally expensive. The original brickmap paper presents a storage scheme that achieves this goal to some extent, as discussed in Section 2.1.3. This chapter will summarize our material storage scheme, extending the brickmap data structure to support material data. Afterward, we discuss how we implemented this new scheme and other key features of our application.

3.1 Technical & user requirements

This project's primary requirements and scope are real-time voxel ray tracing performance for a broad range of hardware and supporting multiple Operating Systems (OSs). Achieving real-time ray tracing fidelity to hardware that would otherwise be limited to traditional rasterization is not trivial. This goal has implications for the technical requirements. The product should be cross-platform and should support at least windows and some Linux distributions with the ability to extend support to macOS in the future. We also want to emphasize that the final product is not an end-user product. The project scope is a rendering API and demo, and the end-user is other programmers.

3.2 Material data and its implication on ray traversal

The *Ray Tracing in One Weekend* book series[12] defines a model for materials on which we base our application materials. There are currently three possible values for a material type: *lambertian*, *metal*, and *dielectric*. Each material corresponds to a Bidirectional Scattering Distribution Function (BSDF). *Lambertian* is a matte or diffuse material, the most common surface type. *Metal* defines a shiny surface that can reflect based on a roughness parameter (fuzz). *Dielectric* can reflect and refract based on different factors. As an example, the dielectric can model water and glass.

The ray tracing algorithm will traverse the brickmap grid, and if a hit is detected, it will perform a lookup. A ray will scatter depending on what material type the hit voxel has.

- **Lambertian** has the simplest behavior. The ray will simply get a direction which is a random vector in the hemisphere relative to its surface normal
- **Metal** will reflect a ray depending on the metal materials *fuzz* value. A metal surface with a fuzz value of 0 will act as a mirror
- **dielectric** can either reflect like metal or refract the ray. Suppose the angle between the ray direction and the surface normal is low. In that case, the ray travels through the material and refracts according to snell's law and the material *internal reflection*. If the ray does not refract, then it will be reflected.

3.3 Storing material information

Storing material information is done by defining virtual buckets. Each bucket has a dedicated slice of a global material index array visible to the GPU. The buckets only have to be used by the CPU. We can check the current bucket of the brick to see if the current bucket size is sufficient when a brick is changed. If the currently assigned bucket of a brick is insufficient, then the next bucket-size free array will contain a bucket with sufficient capacity. We define bucket sizes by the power of 2 and are configurable under application initialization. The biggest buckets are always the max voxel count for a given brick. The biggest buckets are therefore $2^9 = 512$. Depending on the configured number of bucket storage instances, the smallest bucket size will be $N = 9 - BucketCount + 1, 2^N = Size$. So if the application defines a *BucketCount* of 4, then the smallest bucket size will be $2^{(9-4+1)} = 2^6 = 64$. The application will have buckets of the following sizes: $64, 128, 256, 512$. Figure 3.1 illustrate this layout visually. We also store buckets in two individual arrays depending on whether they are used by any brick or free. By using two arrays, we get $0(1)$ performance when we want to allocate a new start index. To free a bucket, we do the following: swap remove the bucket from the occupied bucket array, followed by appending the removed bucket to the free array. A simple pop operation on the free array and append on the occupied array is assigning a bucket to a brick. We also discuss further improvements in Section 5.2.8

As discussed in Section 3.2, there are currently three possible values for a material type: lambertian, metal, and dielectric. All material entries are 16-bit integers. The 2 least significant bits define the type of the material where $0 = lambertian, 1 = metal, 2 = dielectric$. The next 6 bits define the index into the material type buffer. For example, if the type of the material is metal, and the value

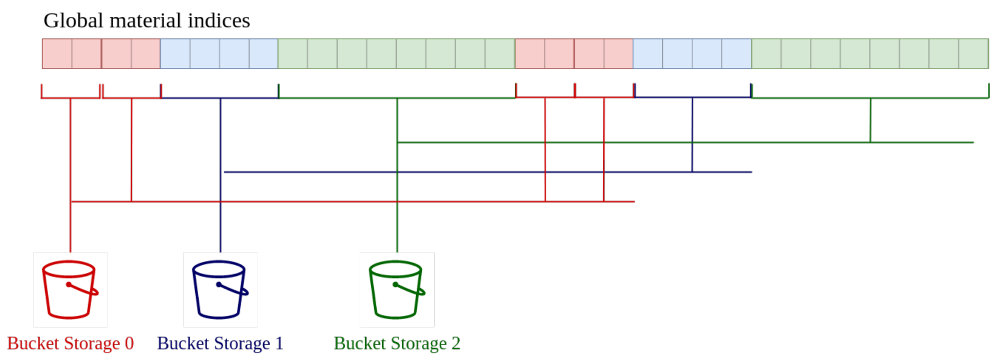


Figure 3.1: Material bucket layout

of the next 6 bits is 32, then the system will look up entry 32 in the metal buffer to find the right properties for the voxel. The final 8 bits index into a color buffer which is the color (RGB) of the material surface. On the GPU, we can extract our 16 bits by doing the following in the GLSL shader:

```
// We store the material data in 16 bit unsigned integers
// but glsl only support 32 bit values so we need
// to extract the relevant 16 bits from a entry pair
const uint material_index = hit.index / 2;
const uint material_bit_offset = 16 * (hit.index % 2);
const uint material_bits = (materials[material_index] >> material_bit_offset);
const Material material = Material(
    material_bits & 3,                    // type
    bitfieldExtract(material_bits, 2, 6),   // material index
    bitfieldExtract(material_bits, 8, 8)    // albedo color
);
```

Code listing 3.1: Extracting material data from bits

3.4 Further memory reduction

Our application use Vulkan which has SPIRV as its target shader language. This project use GLSL to program shaders. The project build execution compiles GLSL to SPIRV and Vulkan loads the compiled SPIRV. GLSL does not support integer widths other than 32bits without extensions which again does not include any guarantees that the target platform support said extension. In our case, the lack of control over integer bit width means that the implemented version of Figure 2.1a takes 32 bits for the LOD index. We can reduce memory usage by using a single index for both the LOD and the color index. GLSL has a built-in `bitfieldExtract` function and support bitwise operations. Both methods can extract information from bits. We can use this to reserve the first bit of an index entry to specify if the index is a start index for voxel color or if the index is a LOD index. By doing so, we reduce the brick memory footprint from 576 to 544

In an application with requirements for a high range of unique colors or unique material properties, an implementation can choose to extend the 16-bit material entries mentioned in Section 3.3 to hold 32 bits per entry. Doubling the size of material entries could introduce issues with memory usage. We can reduce memory usage in an application by limiting unique material data to 255. Two buffers are used—a brickmap start-index index into a material index buffer. The material index buffer is a collection of 8-bit unsigned integers. These 8-bit integers are then indices to the material buffer where the 16/32 bit material data is stored. An intermediate index buffer means that each voxel material index is, in practice, almost halved from 16 or 32 bits per voxel to 8 bits. The storage solution is illustrated in Figure 3.2

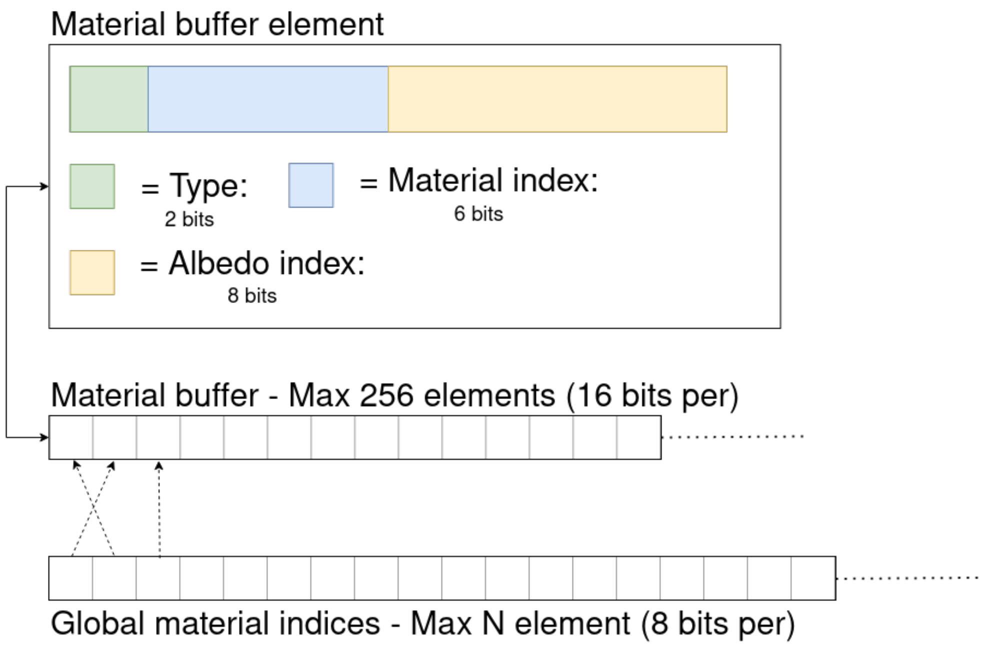

Material buffer element
= Type:
2 bits
= Material index:
6 bits
= Albedo index:
8 bits
Material buffer - Max 256 elements (16 bits per)
Global material indices - Max N element (8 bits per)

Figure 3.2: Material data storage scheme

3.4.1 Parallelization and contingency avoidance

The codebase uses an abstraction called a *grid* to handle any changes to the brickmap data. The grid holds all grid-related data on the host side but uses another abstraction called *workers* to work on the grid efficiently. The workers are the entities that perform any change on the grid, and a grid worker corresponds to one thread on the system.

In order to avoid contingency between threads, each worker thread has a unique range of global material indices. Each worker initializes their bucket pool to request index data without any lock mechanism. The workers have the responsibility to perform any work on the grid. Any thread can request an insert or remove on the voxel grid. The worker will receive a job description scheduled by being inserted into a job queue. By segmenting material index data based on workers, we can perform lockless grid manipulation and avoid any potential contingency. The mentioned index segmentation is illustrated in Figure 3.3

The index segmentation does have some downsides, one of which we discuss more in Section 5.2.8. Another is that neighboring bricks near the border of two worker areas in the grid will most likely not have sequential material data. Non-sequential data can cause a noticeable reduction in render speed when rays have to fetch more memory than usual. The result is that the solution delivers faster grid editing at the cost of reducing the final frame rate when viewing certain parts of the grid.

The scheduling of a job is done with the following lines of zig code:

```zig
/// Asynchrounsly (thread safe) insert a brick at coordinate x y z
fn insert(self: *BrickGrid, x: usize, y: usize, z: usize, material_index: u8)void{
    // find the worker that should be assigned insert
    const worker_index = blk: {
        const grid_x = x / 8;
        // work_segment_size = brick_count_x / workers_count,
        break :blk grid_x / self.state.work_segment_size;
    };

    self.workers[worker_index].registerJob(Worker.Job{ .insert = .{
        .x = x,
        .y = y,
        .z = z,
        .material_index = material_index,
    } });
}
```

Code listing 3.2: Schedule insert to a worker thread

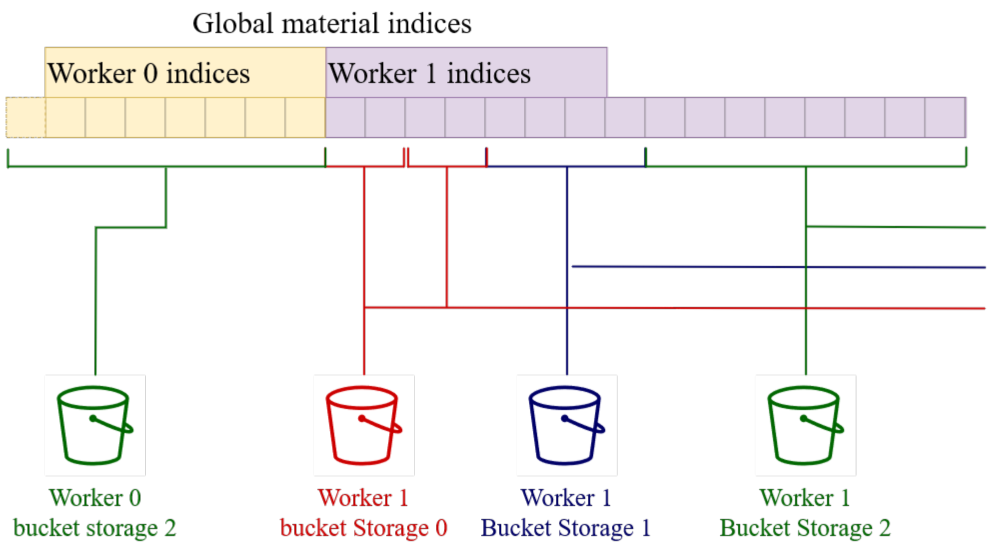


Figure 3.3: Bucket segmentation to avoid thread contingency

3.4.2 Stable transfers

Streaming of data between GPU and CPU is important for the original brickmap's systems like the LOD system. Also, any game system that includes manipulating the voxel grid requires transfers to be seamless and fast. To enable a stable framerate while the grid receives continuous updates, we introduce two systems that alleviate transfer overhead: *DeviceDataDelta* and *StagingRamp*.

Recording data delta for transfer

DeviceDataDelta is a simple abstraction that defines a sub slice of a given data set that has changed since the last transfer. We can use this data to avoid sending existing data repeatedly and reduce the bandwidth. The following listing shows the delta struct state:

```zig
pub const DeviceDataDelta = struct {
    const DeltaState = enum {
        invalid,
        inactive,
        active,
    };

    mutex: Mutex,
    state: DeltaState,
    from: usize,
    to: usize,
}
```

Code listing 3.3: DeviceDataDelta structure

Then there are two function to summaries the delta's API:

```zig
// called while mutex is locked externally
pub fn resetDelta(self: *DeviceDataDelta) void {
    self.state = .inactive;
    self.from = std.math.maxInt(usize);
    self.to = std.math.minInt(usize);
}

pub fn registerDelta(self: *DeviceDataDelta, delta_index: usize) void {
    self.mutex.lock();
    // defer calls expressions at the end of current scope
    defer self.mutex.unlock();

    self.state = .active;
    self.from = std.math.min(self.from, delta_index);
    self.to = std.math.max(self.to, delta_index + 1);
}
```

Code listing 3.4: DeviceDataDelta core API

The delta system has to be cheap both computationally and on memory usage. The behavior used is rudimentary in how it records changes. The recorded delta range is sufficient as changes are usually performed sequentially on the data to utilize the cache. We also further improve the delta by giving each worker thread unique instances of *DeviceDataDelta* when applicable. An example of such data is the bucket segmented material indices as discussed in Section 3.4.1. We give each thread unique material indices *DeviceDataDelta* for this reason.

Before we perform a draw call, we can read each *DeviceDataDelta* and request a transfer to the GPU based on the registered ranges, which are relative to specific buffers. The *StagingRamp* transfers data from the host to the device, which is the next step.

Efficient vulkan buffer transfers

Like everything else in Vulkan, the logic of transferring data is explicit. The application controls most transfer logic manually, so we need to consider how to effectively transfer data without causing noticeable stuttering in the rendering loop. *StagingRamp* abstraction helps eliminate transfer overhead. The *StagingRamp* structure consist of $N, N \geq 1$ *StagingBuffers* which are visible to both the CPU and the GPU. The system schedules transfer jobs using these buffers. However, the actual flushing of data from the host (CPU) to the device (GPU) is performed only once per frame and not for each transfer request. Then a command buffer is executed to perform a copy from the *StagingBuffers* to the destination buffer specified when the transfer was requested.

The *StagingRamp* abstraction will usually try to fill one buffer at a time, but will have to perform fallback for different scenarios when a given buffer is not available as seen in the Listing 3.5

We can see from the listing that we will get the first buffer that is sufficient in size and idle (not currently performing a flush or executing a command buffer). The function will return a `error.StagingBuffersFull` if all buffers are full. Errors are a first-class citizen in Zig. Zig is the chosen programming language for this project, as discussed in Section 3.8.1. When we call the function listed above, we can pattern match the returned value and check if the value is the mentioned `error.StagingBuffersFull`. If we get this error, we can cache the transfer request and defer it to the next frame transfer operation.

```
inline fn getIdleRampIndex(self: *StagingRamp, ctx: Context, size: vk.DeviceSize) !
    usize {
    var full_ramps: usize = 0;
    // get a idle buffer
    var index: usize = blk: {
        for (self.staging_ramps) |ramp, i| {
            // if ramp is out of memory
            if (ramp.buffer_cursor + size >= buffer_size) {
                full_ramps += 1;
                continue;
            }
            // if ramp is idle
            if ((try ctx.vkd.getFenceStatus(ramp.fence)) == .success) {
                // get the index
                break :blk i;
            }
        }
        break :blk (self.last_buffer_used + 1) % self.staging_ramps.len;
    };
    // if none of the ramps has sufficient storage for the requested transfer
    if (full_ramps >= self.staging_ramps.len) {
        return error.StagingBuffersFull;
    }
    defer self.last_buffer_used = index;

    // wait for previous transfer (in case all ramps were busy)
    _ = try ctx.vkd.waitForFences(...buffers[index].fence); // some arguments
        emitted

    return index;
}
```

Code listing 3.5: Requesting index of an idle staging buffer

3.5 Sun and shadows in the application

Optimized light sampling is not a trivial topic. There have been great strides in light sampling in recent years. ReSTIR and its successors are one example of effective multiple light source sampling[13]. Utilizing such techniques takes substantial development time and was therefore not included in the project. Given the time restrictions, this project does not implement arbitrary light sources in the grid and optimized sampling of such lights.

We do include a single *sun* light source. The sun is an arbitrary disc not restricted to the voxel grid. It has a light color, position, and radius. When enabled in the scene, each ray hit will fire a subsequent *shadow ray* that will attempt to hit the defined sun disc. If the shadow ray hits the sun, the color at that point will be a combination of the sun's emitted light color and the surface color attenuation. Extending shadow rays so that a hit will sample nearby light sources to make more complex light interactions is possible. However, making it performant for real-time rendering would require significant consideration.

The sun object can also be animated using the developer GUI. By enabling animation on the sun, it will rotate as expected around the voxel grid much as the earth rotates around the actual sun. The animation procedure is simple. The sun is initially placed at an arbitrary distance from the grid center. Three or more orientations are defined using quaternion math. After these initial steps, a spherical linear interpolation is performed between the three or more previously mentioned quaternions to get a smooth rotation around the grid over time. The sun's color can change similarly to the quaternion orientations throughout a solar cycle. Colors converge using lerp during the same procedure as the quaternion slerp. For example, in the benchmark scene discussed in Section 4.4 color changes to red during dusk and dawn and a yellow color when the sun is at its highest point.

3.6 Terrain generation for procedural scenes

In order to perform any inspection on ray accuracy and performance, we need scenes to ray trace. We procedurally generate some simple scenes since voxel data is not necessarily standardized as polygon-based mesh data. We generate primitive terrain by utilizing Perlin noise which is a gradient noise. We generate a height value using the Perlin noise for each x, y point in the grid. The result of this procedure is a height map. As long as the random generator is given the same seed, we get the same height map. When the system has calculated the height for a given x, y value, it will dispatch a list of insert jobs to the grid thread/workers. Height will affect the voxel type, so a voxel at a lower height will likely be assigned dirt or grass material, while higher voxels become stone or dirt. This job is performed on multiple cores on the CPU (multithreaded).

So far, all mentioned materials are categorized as lambertian/diffuse materials. After the initial terrain generation step, the terrain generation will spawn ”water” voxels. Water voxels add complexity to the scene since the dielectric material enables rays to reflect and refract. The water voxels are all under a runtime specified *water height* value and are unassigned after the initial terrain generation. *Water* is a dielectric material type with a blue color.

3.7 Benchmark module

We made a benchmark module in our code base. The purpose of the benchmark is not to compare our method with other methods but to compare the application performance with different hardware and application configurations. The performance of the application is highly dependent on where the camera is in the scene because different materials and geometry will affect how many times the ray bounces and how expensive the scatter computation will be, as discussed in Section 3.2.

The role of the benchmarking module is simple:

1. Define a predefined camera path where both orientation and position are affected
2. Apply the camera path over a given time. The time it takes to complete the path must be frame time-independent.
3. Generate a report on the application performance throughout benchmark run time. It also includes information about hardware and current scene configuration.

Initializing the benchmark ensures that the sunlight is in the same start position as previous benchmarking sessions since frame times are affected by sun position when enabled.

We also integrate this benchmarking tool with the GUI which we describe in Section 3.8.4. We can trigger a benchmark to test different scene configurations without restarting the application. The generated reports are used in Section 4.4 to gather data.

3.8 Tools

Tool is a broad term that covers many different topics in a development cycle. Some tools are essential for the project's success others are used to speed up development. An example of an essential tool is the chosen programming language(s). In the context of our application, the performance overhead of some languages is too significant to be practical. Interpreted languages like Python would cause issues to the performance even with the correct use of algorithms and careful consideration of the whole program's semantics performance.

We cover the tools used to make it easier to reproduce our results, whether it is final performance or a simple collection of performance analytics.

3.8.1 Zig

The chosen programming language for the project ended up being Zig. Zig is a new Work In Progress (WIP) programming language and toolchain. Zig is a toolchain because it can build traditional C or C++ projects, much like building tools such as CMake or ninja. Zig supports this feature because Zig has a solid Foreign Function Interface (FFI) with C and can manage mixed projects with Zig and C/C++ code. Zig is general purpose and aims to be low level with manual memory management and high performance out of the box.

The mentioned FFI capabilities are essential for a rendering application since we will need to interface with specific libraries. Vulkan is one such library that is written in C and has to be dynamically linked. Zig also has a limited number of libraries written in pure Zig. Some libraries require years of testing and development before they are ready for external usage, for example, cross-platform window management with support for graphics API interfacing. Luckily, this is not a problem because the FFI allows us to interface with classic C APIs without much hassle.

Unit tests

Tests are first-class citizens in Zig, and Zig can run tests manually file by file. Usually, a Zig project links all tests to a single `zig build test` command by utilizing a Zig build script. Tests can be defined anywhere in a zig project, but a given test is preferably close to the tested code (same file or folder). Tests are defined by writing `test` followed by a compile-time string which will be the name displayed by the output.

We used test-driven development throughout the early phase of the project. We got issues with doing test-driven for certain aspects of the rendering code. Sections of the rendering system are intertwined, and testing Vulkan calls were deemed excessive. Currently, some modules in the codebase are unit tested.

We did our best to test code that was not directly related to interfacing with Vulkan. An example of a unit test is included in the listing below, and the code is from the vox model loader we discuss in Section 3.8.6.

```zig
// here we see the name of the test which will be printed in the test report
test "validateHeader: invalid version detected" {
    // currently 150 is the only valid version in vox,
    // here it is 123 and should fail on parsing.
    // "++" means compile time concatenation of arrays.
    const invalid_test_buffer: []const u8 = "VOX " ++ [_]u8{ 123, 0, 0, 0 } ++ "
        MAIN";

    try std.testing.expectError(
        ParseError.UnexpectedVersion,
        validateHeader(invalid_test_buffer[0..]),
    );
}
```

Code listing 3.6: Unit test in vox loader module

3.8.2 Vulkan

Vulkan is a graphics API much like OpenGL, Metal and DirectX. Graphics APIs serve as an intermediate layer between hardware drivers and the application in order to unify the interface to the system GPU(s). Vulkan is a cross-platform open-source API that aims to be high performance and conform more to modern hardware compared to its predecessor OpenGL. We chose to use Vulkan for this project to control the rendering behavior better and enable better performance. The cost of this performance is verbosity and increased development friction. The project uses Vulkan to perform GPGPU compute for ray tracing generation and traversal and to draw the resulting image to a window context.

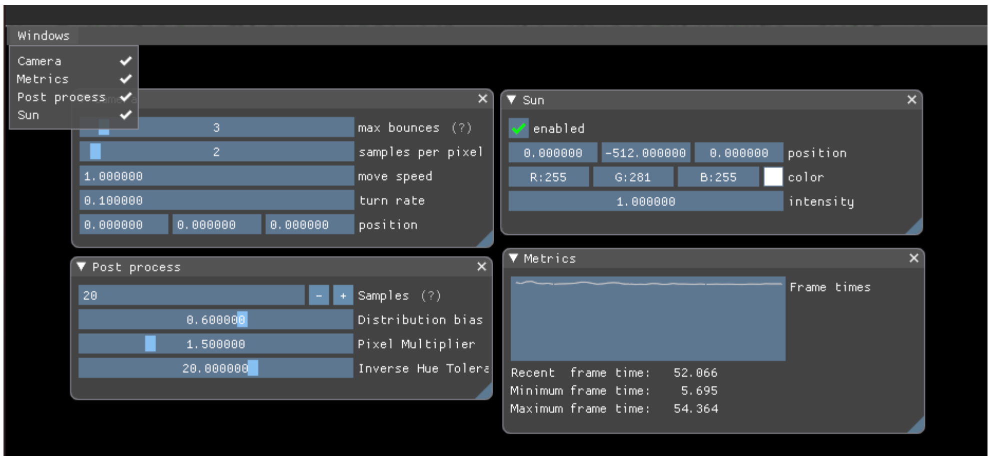
Vulkan also introduces the concept of validation layers that can intercept Vulkan API calls and validate if the use of the API is specification compliant. These validation layers can be conducive to avoiding common pitfalls and detecting bugs. A third-party developer and hardware vendors can implement custom layers to catch bad practices or problematic usage.

3.8.3 GLFW

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES, and Vulkan development on the desktop. It provides a simple API for creating windows, contexts, and surfaces, receiving input and events.[14]. There are also Zig bindings made for the library, which makes it easier to integrate the library functions with some of Zig's new features like error handling. The project uses GLFW to receive input events from the OS and to open and deal with application windows in a platform-agnostic way.

3.8.4 Imgui

Imgui is a immediate mode Graphical User Interface (GUI) library that is agnostic to most graphical APIs. It accelerates the development of GUI for developers or end-users. GUI is helpful to enhance the control of the application parameters without needing to restart the application or recompile. Figure 3.4 illustrate example application parameters.



```
3                                   max bounces (?)
2                                   samples per pixel
1.000000                            move speed
0.100000                            turn rate
0.000000   0.000000   0.000000      position
```

```
✓ enabled
0.000000   -512.000000   0.000000   position
R:255      G:281         B:255      color
1.000000                            intensity
```

```
20                        - +       Samples (?)
0.600000                            Distribution bias
1.500000                            Pixel Multiplier
20.000000                           Inverse Hue Tolera
```

```
Recent  frame time:   52.066
Minimum frame time:    5.695
Maximum frame time:   54.364
```

Figure 3.4: Imgui based developer menus

3.8.5 Profiling & debugging

Profiling is quintessential when developing performance-critical software. Making software effective is an iterative process, and being able to locate hotspots in the software empirically is needed. Debugging is also a challenge, especially for GPU based software, as the GPU execution is generally more difficult to observe compared to CPU execution.

Tracy

Tracy is a profiling tool described by the author as *a real-time, nanosecond resolution, remote telemetry, hybrid frame and sampling profiler for games and other* *applications*[15]. Figure 3.5 display a profile session. Tracy includes two modules, a server and a Application Programming Interface (API). The API can mark functions, threads, and more. It mainly supports profiling C/C++ and Lua, but subsequently, all languages with FFI with C, which includes Zig as discussed in Section 3.8.1. The explicit marking of code through the API allows for high precision measurements and detailed information about a given frame or thread. The tool also supports GPU profiling for applications using the Vulkan API.

The server module is the main method for intercepting marked zones. It also includes a user GUI that visualizes the application game loop and all marked frames, function frames, and threads. The GUI is illustrated in Figure 3.5. Tracy also supports a Command Line Interface (CLI) that allows you to gather profile data as files. The Tracy server GUI can inspect the Tracy files.

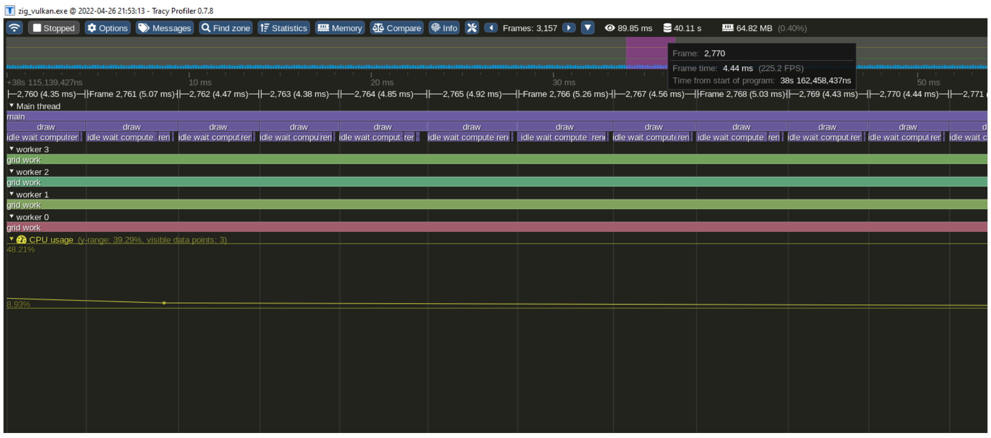


Figure 3.5: Tracy profile session example

Renderdoc

RenderDoc is a free MIT licensed stand-alone graphics debugger that allows quick and easy single-frame capture and detailed introspection of any application using Vulkan ...[16]. It is a beneficial tool that gives insight about GPU buffer memory annotated by the variable names from shader source files, textures used by the rendering and compute-pipelines, debugging of compute threads (although an experimental feature), and much more. Renderdoc can also be used in conjunction with Vulkans ability to print from the GPU to inspect messages from a given shader with the ability to search and filter on thread ids and message content.

Radeon™ Developer Tool Suite.

For AMD hardware profiling, the Radeon GPU Profiler (RGP) and Radeon GPU Analyzer (RGA) tool kit was used on Linux. AMD also offers a tool to analyze memory layout and usage (Radeon Memory Analyzer (RMA)), but it only supports Windows and has not been used during development. RGP is a *low-level* *optimization tool that provides detailed information on Radeon™ GPU*[17]. It offers insight into GPU behaviour according to API usage. Synchronization primitives can be inspected to verify that they are used in a manner that does not introduce redundant overhead. GPU queues can be inspected to see the execution of individual command buffers. There are many more features like inspection of wavefront occupancy and cache misses.

RGA *is an offline compiler and performance analysis tool for DirectX®, Vulkan®, SPIR-V™, OpenGL®, and OpenCL™*[17]. It offers insight into the executed GPU instructions and provides profiles on the instructions in order to identify shader hotspots. There is no shader to instruction correlation feature for Vulkan yet, so the user has to familiarize oneself with AMD's RDNA 2 Instruction Set Architecture (ISA), or the relevant ISA for currently used Graphics Processing Unit (GPU). Documentation for the RDNA ISA is easily accessible so that the correlation can be performed manually with some practice[18].

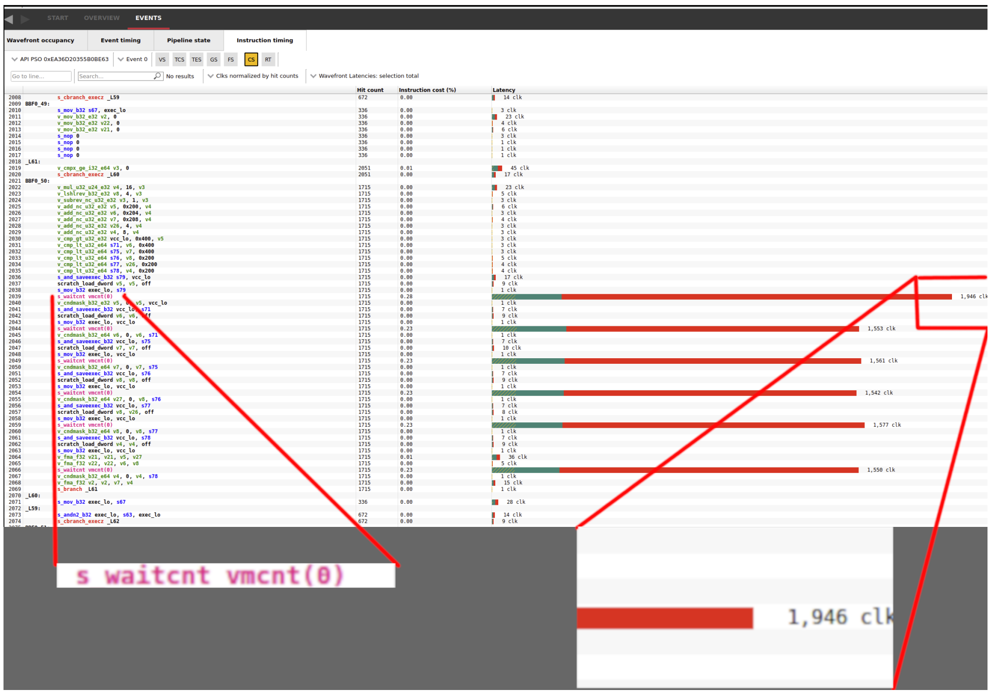
Nsight toolkit

We utilized Nsight Graphics and Nsight Systems for Nvidia hardware profiling. The feature set of these two applications is similar to features offered by the AMD toolkit, as discussed in Section 3.8.5. Nsight Systems is a performance analysis tool that allows insight into most hardware usage, including memory, CPU and GPU workload. Nsight Systems is helpful for *more comprehensive* profiling and detecting slow areas in the application. Nsight Graphics gives insight into the GPU workload. Using this tool, we can analyze GPU memory usage, operations, and shader profiling to identify performance issue hotspots in the GPU.

Profiling session example with RGA

Figure 3.6 illustrates an initial shader profile. We can see that some instructions take up to 2000 clocks. The green section of these hotspots indicates the latency of instruction being hidden by the VALU (Vector Arithmetic-Logic Unit (ALU)), and a combined green-yellow means hidden by both VALU and SALU (Scalar ALU). Red means clocks that are stalling by being idle. Clock latency is marked as hidden by either VALU or SALU, which means that latency is not detrimental as long as the bar is mostly **not** red. In the profile mentioned in Figure 3.6, we can see the bar is mostly red. All the hotspots in the current profile is called *s_waitcmt_vmcnt* which means: *Wait for the counts of outstanding vector memory events*[18]. We can conclude that the buffers used are not suitable for the applications.

Luckily two articles have the relevant information to improve memory read efficiency. Nvidia engineers wrote the first article titled: *Vulkan Memory Management*[19]. They describe how an application should strive towards significant allocations and split Vulkan buffers and memory ”virtually” in the application into sub-allocations when needed, called pooling. Pooling means that more memory can be sequential and is friendlier to the cache. Pooling was not used in the applic-



```
                                              Hit count  Instruction cost (%)  Latency
            s_cbranch_execz _L59              672        0.00                  14 clk
BBF0_49:
            s_mov_b32 s67, exec_lo            336        0.00                  3 clk
            v_mov_b32_e32 v2, 0               336        0.00                  23 clk
            v_mov_b32_e32 v22, 0              336        0.00                  4 clk
            v_mov_b32_e32 v21, 0              336        0.00                  6 clk
            s_nop 0                           336        0.00                  3 clk
            s_nop 0                           336        0.00                  1 clk
            s_nop 0                           336        0.00                  1 clk
            s_nop 0                           336        0.00                  1 clk
_L61:
            v_cmpx_ge_i32_e64 v3, 0           2051       0.01                  45 clk
            s_cbranch_execz _L60              2051       0.00                  17 clk
BBF0_50:
            v_mul_u32_u24_e32 v4, 16, v3      1715       0.00                  23 clk
            v_lshlrev_b32_e32 v8, 4, v3       1715       0.00                  5 clk
            v_subrev_nc_u32_e32 v3, 1, v3     1715       0.00                  3 clk
            v_add_nc_u32_e32 v5, 0x200, v4    1715       0.00                  6 clk
            v_add_nc_u32_e32 v6, 0x204, v4    1715       0.00                  3 clk
            v_add_nc_u32_e32 v7, 0x208, v4    1715       0.00                  4 clk
            v_add_nc_u32_e32 v26, 4, v4       1715       0.00                  3 clk
            v_add_nc_u32_e32 v4, 8, v4        1715       0.00                  3 clk
            v_cmp_gt_u32_e32 vcc_lo, 0x400, v5   1715    0.00                  3 clk
            v_cmp_lt_u32_e64 s71, v6, 0x400   1715       0.00                  3 clk
            v_cmp_lt_u32_e64 s75, v7, 0x400   1715       0.00                  3 clk
            v_cmp_lt_u32_e64 s76, v8, 0x200   1715       0.00                  5 clk
            v_cmp_lt_u32_e64 s77, v26, 0x200  1715       0.00                  4 clk
            v_cmp_lt_u32_e64 s78, v4, 0x200   1715       0.00                  4 clk
            s_and_saveexec_b32 s79, vcc_lo    1715       0.00                  17 clk
            scratch_load_dword v5, v5, off    1715       0.00                  9 clk
            s_mov_b32 exec_lo, s79            1715       0.00                  1 clk
            s_waitcnt vmcnt(0)                1715       0.28                  1,946 clk
            v_cndmask_b32_e32 v5, 0, v5, vcc_lo  1715    0.00                  1 clk
            s_and_saveexec_b32 vcc_lo, s71    1715       0.00                  7 clk
            scratch_load_dword v6, v6, off    1715       0.00                  9 clk
            s_mov_b32 exec_lo, vcc_lo         1715       0.00                  1 clk
            s_waitcnt vmcnt(0)                1715       0.23                  1,553 clk
            v_cndmask_b32_e64 v6, 0, v6, s71  1715       0.00                  1 clk
            s_and_saveexec_b32 vcc_lo, s75    1715       0.00                  7 clk
            scratch_load_dword v7, v7, off    1715       0.00                  10 clk
            s_mov_b32 exec_lo, vcc_lo         1715       0.00                  1 clk
            s_waitcnt vmcnt(0)                1715       0.23                  1,561 clk
            v_cndmask_b32_e64 v7, 0, v7, s75  1715       0.00                  1 clk
            s_and_saveexec_b32 vcc_lo, s76    1715       0.00                  7 clk
            scratch_load_dword v8, v8, off    1715       0.00                  9 clk
            s_mov_b32 exec_lo, vcc_lo         1715       0.00                  1 clk
            s_waitcnt vmcnt(0)                1715       0.23                  1,542 clk
            v_cndmask_b32_e64 v27, 0, v8, s76 1715       0.00                  1 clk
            s_and_saveexec_b32 vcc_lo, s77    1715       0.00                  7 clk
            scratch_load_dword v8, v26, off   1715       0.00                  8 clk
            s_mov_b32 exec_lo, vcc_lo         1715       0.00                  1 clk
            s_waitcnt vmcnt(0)                1715       0.23                  1,577 clk
            v_cndmask_b32_e64 v8, 0, v8, s77  1715       0.00                  1 clk
            s_and_saveexec_b32 vcc_lo, s78    1715       0.00                  7 clk
            scratch_load_dword v4, v4, off    1715       0.00                  9 clk
            s_mov_b32 exec_lo, vcc_lo         1715       0.00                  1 clk
            v_fma_f32 v21, v21, v5, v27       1715       0.01                  36 clk
            v_fma_f32 v22, v22, v6, v8        1715       0.00                  5 clk
            s_waitcnt vmcnt(0)                1715       0.23                  1,550 clk
            v_cndmask_b32_e64 v4, 0, v4, s78  1715       0.00                  1 clk
            v_fma_f32 v2, v2, v7, v4          1715       0.00                  15 clk
            s_branch _L61                     1715       0.00                  1 clk
_L60:
            s_mov_b32 exec_lo, s67            336        0.00                  28 clk
_L59:
            s_andn2_b32 exec_lo, s63, exec_lo 672        0.00                  14 clk
            s_cbranch_execz _L62              672        0.00                  9 clk
```

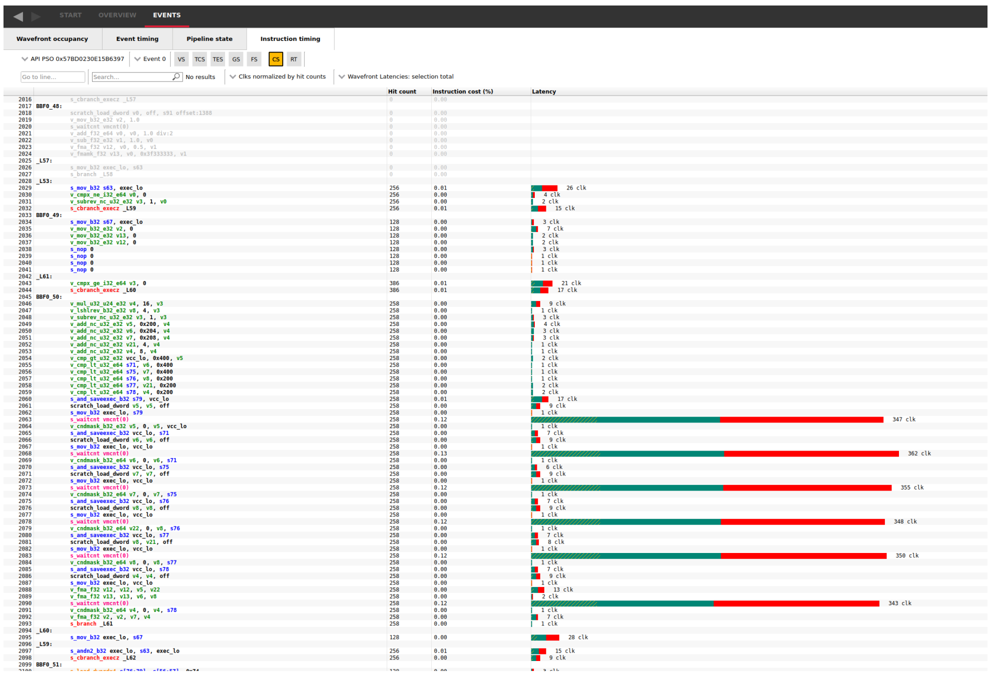
Figure 3.6: Initial Radeon GPU Analyzer profile (red means idle, green indicates that the GPU is busy)

ation when capturing the profile illustrated by Figure 3.6, so this was an obvious optimization path.

The second article is written by an AMD engineer and has the title: *Using Vulkan® Device Memory*[6]. This article describes how different memory configurations can be categorized into heap types from heap 0 to 2. The fastest memory category for GPU operations is heap 0, which can only be manipulated and read by the GPU itself. Heap 1 is the second fastest and has the potential to be modified by the CPU. Heap 2 is an even slower category visible to the CPU, and the graphics driver is responsible for memory consistency between the host and the device. In our application, all the memory was under the heap 2 category.

With the new knowledge about our shader hotspots and optimal Vulkan memory usage, we can apply two significant changes to how our compute pipeline use GPU memory. Previously we used unique buffers and memory for each shader buffer. We change this only to allocate one Vulkan buffer, and memory location shared between uniform and storage data. We split this data virtually and bind different buffer sections to our declared shader buffers. We also change the buffer type to be GPU local and make the required changes to categorize this memory from heap 2 to heap 0. As a heap 0 buffer, the host cannot upload data to this buffer directly, but we can introduce a new *staging buffer* that is categorized as heap 1, which is accessible from the host. We can then upload our main memory to the staging buffer and perform a copy operation on the GPU, so relevant segments of staging buffer memory are copied to the correct location in our main memory.

After applying these changes, a new profile can be collected, illustrated in Figure 3.7. We can see a considerable improvement in clock usage. The worst instruction on line 2039 in Figure 3.6 went from **1946** to **347** clocks, and we see that the ratio of the red section on the bar is reduced. The *s_waitcmt_vmcnt* instruction also has a reduction in overall clock usage of around 70% after we have applied the changes to our Vulkan buffer(s).



```
                                                         Hit count  Instruction cost (%)  Latency
2016          s_cbranch_execz _L57                       0          0.00
2017 BBF0_48:
2018          scratch_load_dword v0, off, s91 offset:1388  0        0.00
2019          v_mov_b32_e32 v2, 1.0                      0          0.00
2020          s_waitcnt vmcnt(0)                         0          0.00
2021          v_add_f32_e64 v0, v0, 1.0 div:2            0          0.00
2022          v_sub_f32_e32 v1, 1.0, v0                  0          0.00
2023          v_fma_f32 v12, v0, 0.5, v1                 0          0.00
2024          v_fmamk_f32 v13, v0, 0x3f333333, v1        0          0.00
2025 _L57:
2026          s_mov_b32 exec_lo, s63                     0          0.00
2027          s_branch _L58                              0          0.00
2028 _L53:
2029          s_mov_b32 s63, exec_lo                     256        0.01                  26 clk
2030          v_cmpx_ne_i32_e64 v0, 0                    256        0.00                  4 clk
2031          v_subrev_nc_u32_e32 v3, 1, v0              256        0.00                  2 clk
2032          s_cbranch_execz _L59                       256        0.01                  15 clk
2033 BBF0_49:
2034          s_mov_b32 s67, exec_lo                     128        0.00                  3 clk
2035          v_mov_b32_e32 v2, 0                        128        0.00                  7 clk
2036          v_mov_b32_e32 v13, 0                       128        0.00                  2 clk
2037          v_mov_b32_e32 v12, 0                       128        0.00                  2 clk
2038          s_nop 0                                    128        0.00                  3 clk
2039          s_nop 0                                    128        0.00                  1 clk
2040          s_nop 0                                    128        0.00                  1 clk
2041          s_nop 0                                    128        0.00                  1 clk
2042 _L61:
2043          v_cmpx_ge_i32_e64 v3, 0                    386        0.01                  21 clk
2044          s_cbranch_execz _L60                       386        0.01                  17 clk
2045 BBF0_50:
2046          v_mul_u32_u24_e32 v4, 16, v3               258        0.00                  9 clk
2047          v_lshlrev_b32_e32 v8, 4, v3                258        0.00                  1 clk
2048          v_subrev_nc_u32_e32 v3, 1, v3              258        0.00                  3 clk
2049          v_add_nc_u32_e32 v5, 0x200, v4             258        0.00                  4 clk
2050          v_add_nc_u32_e32 v6, 0x204, v4             258        0.00                  3 clk
2051          v_add_nc_u32_e32 v7, 0x208, v4             258        0.00                  3 clk
2052          v_add_nc_u32_e32 v21, 4, v4                258        0.00                  1 clk
2053          v_add_nc_u32_e32 v4, 8, v4                 258        0.00                  1 clk
2054          v_cmp_gt_u32_e32 vcc_lo, 0x400, v5         258        0.00                  2 clk
2055          v_cmp_lt_u32_e64 s71, v6, 0x400            258        0.00                  1 clk
2056          v_cmp_lt_u32_e64 s75, v7, 0x400            258        0.00                  1 clk
2057          v_cmp_lt_u32_e64 s76, v8, 0x200            258        0.00                  1 clk
2058          v_cmp_lt_u32_e64 s77, v21, 0x200           258        0.00                  2 clk
2059          v_cmp_lt_u32_e64 s78, v4, 0x200            258        0.00                  2 clk
2060          s_and_saveexec_b32 s79, vcc_lo             258        0.01                  17 clk
2061          scratch_load_dword v5, v5, off             258        0.00                  9 clk
2062          s_mov_b32 exec_lo, s79                     258        0.00                  1 clk
2063          s_waitcnt vmcnt(0)                         258        0.12                  347 clk
2064          v_cndmask_b32_e32 v5, 0, v5, vcc_lo        258        0.00                  1 clk
2065          s_and_saveexec_b32 vcc_lo, s71             258        0.00                  7 clk
2066          scratch_load_dword v6, v6, off             258        0.00                  9 clk
2067          s_mov_b32 exec_lo, vcc_lo                  258        0.00                  1 clk
2068          s_waitcnt vmcnt(0)                         258        0.13                  362 clk
2069          v_cndmask_b32_e64 v6, 0, v6, s71           258        0.00                  1 clk
2070          s_and_saveexec_b32 vcc_lo, s75             258        0.00                  6 clk
2071          scratch_load_dword v7, v7, off             258        0.00                  9 clk
2072          s_mov_b32 exec_lo, vcc_lo                  258        0.00                  1 clk
2073          s_waitcnt vmcnt(0)                         258        0.12                  355 clk
2074          v_cndmask_b32_e64 v7, 0, v7, s75           258        0.00                  1 clk
2075          s_and_saveexec_b32 vcc_lo, s76             258        0.00                  7 clk
2076          scratch_load_dword v8, v8, off             258        0.00                  9 clk
2077          s_mov_b32 exec_lo, vcc_lo                  258        0.00                  1 clk
2078          s_waitcnt vmcnt(0)                         258        0.12                  348 clk
2079          v_cndmask_b32_e64 v22, 0, v8, s76          258        0.00                  1 clk
2080          s_and_saveexec_b32 vcc_lo, s77             258        0.00                  7 clk
2081          scratch_load_dword v8, v21, off            258        0.00                  8 clk
2082          s_mov_b32 exec_lo, vcc_lo                  258        0.00                  1 clk
2083          s_waitcnt vmcnt(0)                         258        0.12                  350 clk
2084          v_cndmask_b32_e64 v8, 0, v8, s77           258        0.00                  1 clk
2085          s_and_saveexec_b32 vcc_lo, s78             258        0.00                  7 clk
2086          scratch_load_dword v4, v4, off             258        0.00                  9 clk
2087          s_mov_b32 exec_lo, vcc_lo                  258        0.00                  1 clk
2088          v_fma_f32 v12, v12, v5, v22                258        0.00                  13 clk
2089          v_fma_f32 v13, v13, v6, v8                 258        0.00                  2 clk
2090          s_waitcnt vmcnt(0)                         258        0.12                  343 clk
2091          v_cndmask_b32_e64 v4, 0, v4, s78           258        0.00                  1 clk
2092          v_fma_f32 v2, v2, v7, v4                   258        0.00                  7 clk
2093          s_branch _L61                              258        0.00                  1 clk
2094 _L60:
2095          s_mov_b32 exec_lo, s67                     128        0.00                  28 clk
2096 _L59:
2097          s_andn2_b32 exec_lo, s63, exec_lo          256        0.01                  15 clk
2098          s_cbranch_execz _L62                       256        0.00                  9 clk
2099 BBF0_51:
```

Figure 3.7: Optimized buffer profile result

3.8.6 MagicaVoxel and the vox file format

MagicaVoxel is an application for voxel modeling, much like blender is for traditional meshes. The application allows the user to make voxel models that can be exported to different formats. One of the formats is the *vox* format that the authors of MagicaVoxel designed. This format is made for voxel data, making it more suited for voxel storage since traditional model formats are not volumetric but surface-based. The application also has a path tracer that can inspect models and is a good reference for comparing our application output images with MagicaVoxel's offline path tracer.

We made a custom vox file loader in the Zig programming language that can be used in conjunction with our ray tracer. We can use the loader to test the rendering system and compare its output with MagicaVoxel's path tracer output. It can also enable more visually exciting scenes in the project renderer. The vox file format specification is open to the public and can therefore be implemented without reverse engineering[20].

3.8.7 IDE: Visual Studio Code + CodeLLDB

We chose Visual Studio Code (Code) as our Integrated Development Environment (IDE). Code is a highly modular IDE with user-generated modules that can be downloaded through an extension marketplace. CodeLLDB was used for debug sessions since it enables breakpoint debugging even in Zig. CodeLLDB integrates features of LLDB to Code. LLDB allows memory inspection in debug binaries which can help during debugging sessions. Zig also has a language server called ZLS (Zig Language Server). There is an extension that integrates ZLS with Code so that the IDE can give syntax highlighting and report bad coding practices relative to idiomatic Zig.

Chapter 4

Discussions and Results

Development is not usually a linear process. Some attempts at progressing development might be unsuitable and might not lead to progress in the project; if an attempt to progress leads to nothing, reflecting on why the attempt was not productive can be constructive. This chapter discusses features that did not improve the application in any meaningful way and, therefore, should have been omitted from the project. In this chapter we also describe benchmarks the prototype application with the new material storage scheme and discuss some of the findings from these benchmarks. Configuration of the ray tracing system will also be discussed and how it might affect performance.

4.1 Performance impact of application configurations

The application can configure the ray tracing step in different ways to achieve desired frame rate at the cost of frame fidelity. We have made sure that the user can try different configurations using the developer GUI in Section 3.8.4 which is illustrated in Figure 3.4. During the runtime, the user can change parameters like:

- Max ray bounces - how many times the ray can bounce before terminating. A higher bounce count leads to more interaction between the lighting and the scene's surfaces.
- Samples per pixel - The application fires 1 to N rays for each pixel. More samples lead to less noise and aliasing in the output image

The application also uses two resolutions which are called *Internal resolution* and *External resolutions*. The *internal resolution* describes the resolution of the ray tracing image, while the *external* describes the resolution of the final output image. The *internal* has bigger implications on the performance of the application since it is likely that any bottlenecks of the application will occur in the ray tracing stage. Each pixel of the *internal resolution* adds computation to the ray tracing step because of the following rule: $InternalPixelCount * SamplesPerPixel = PrimaryRayCount$.

A large cave will be more computationally expensive to render than a flat open field because the max ray bounce is more likely to occur. There is also a varying degree of performance depending on the camera's position in the scene. The application and scene configuration must be identical to perform valid hardware and code benchmarking.

4.2 Poor bucket layout

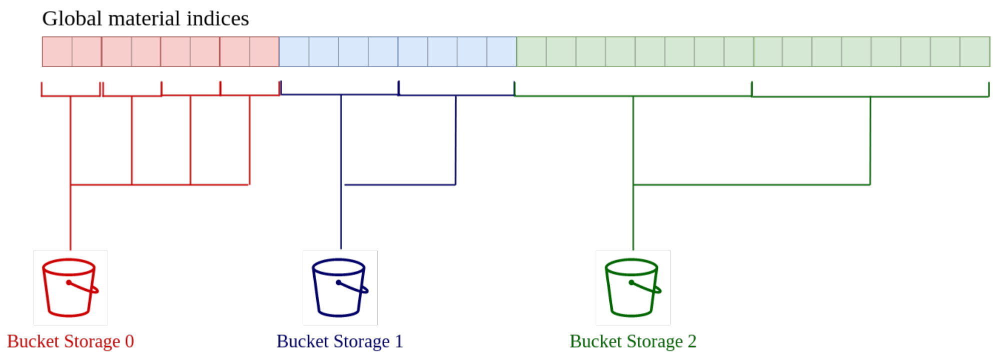


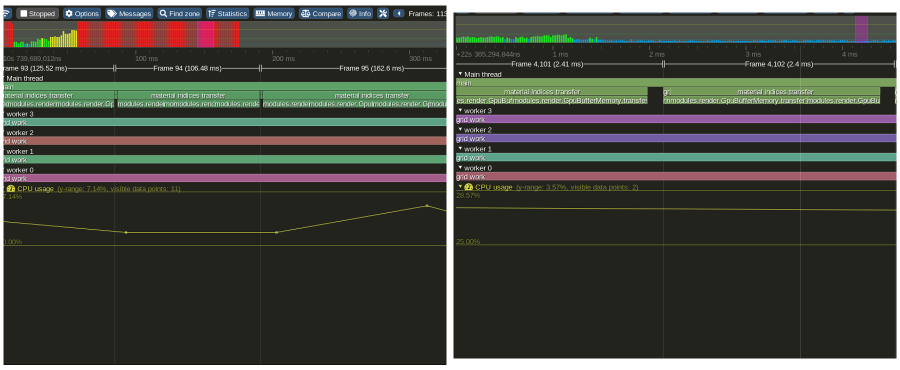
Figure 4.1: Naive bucket layout which has greater probability of cache miss

The initial implementation of the bucket storage presented in Section 3.3 was different from the layout in Figure 3.1. Each bucket had its dedicated segment in the material index array, illustrated in Figure 4.1. The old layout caused major cache coherence issues, which manifested in poor performance as rays invalidated the current cache when they attempted to access material information of a brick. When a brick is assigned bucket size N, and the neighbor brick bucket size $N+X$ (or $N-X$), the cache is evicted because the average byte distance between a given N bucket and $N+1$ bucket is much greater than the byte distance of N bucket and $N+1$ in the current implementation. A ray tends to bounce and access different brick material data in each frame, so the closer each start index is, the greater the chance that memory is cached.

It also caused issues with our delta range system, which we discussed in Section 3.4.2. In Figure 4.2 we can see the before and after we applied the changes to bucket layout in the context of transfer overhead for frame times. Note that the performance shown in this figure was before we applied buffer changes discussed in Section 3.4.2 which has further improved frame times while transferring large chunks of data.

4.3 Suboptimal optimizations

Some attempts at optimizing the application failed. These optimizations are included so that others can avoid spending time implementing these features. These



(a) Poor bucket layout causing transfers to take +100ms of frame time

(b) Good bucket layout allowing delta transfers system to optimize transfers

Figure 4.2: Tracy profile before and after improved bucket layout

optimizations might be valid for other applications. However, ray tracing the brickmap data structure does not yield performance gains with these techniques that are worth the added code complexity of these features.

4.3.1 Ray tracing with Asynchronous Compute

Asynchronous computing has improved execution times for compute workloads bound by instructions as discussed in Section 2.2.4. The project codebase support asynchronous compute and it can be enabled through some simple inline configurations. Some preliminary testing showed performance degradation when using asynchronous compute. Some cards were affected more than others. The RX 6800 XT suffered minor degradation, while the RTX Titan had double frame times.

The frame times also had large oscillations. We believe memory issues cause the oscillations and the performance degradation. The ray tracing step is already bottlenecked by memory retrieve with synchronous compute. Section 2.2.4 mentioned how workloads bottlenecked by memory access might not have a good performance increase with asynchronous compute. In our case, cache coherence might suffer since the GPU will be multitasking more than one workload, which will increase the required amount of local caches.

4.3.2 Shared memory for grid

Nvidia cards were profiled because of their performance difference compared to the AMD 6800 XT. The performance difference is discussed in Section 4.4.1. Profile data was gathered using nsight. The nsight toolkit is discussed in Section 3.8.5. The biggest hotspot on the RTX Titan was identified as the code section that performs the grid buffer's read. Grid access is when the raytracer checks if the brick is empty or not to identify if further lookups are required.

GLSL shared memory was used in order to attempt to optimize the reading of this buffer by loading the buffer into the work group (thread block) local cache.

Shared memory works by explicitly localizing the memory closer to each thread by loading it into the block's cache. The storage buffer data must be moved into the shared memory region, and then the shared memory can be used instead of the storage buffer. Further benchmarks of shared memory showed minimal improvements on smaller grids and reduced performance on larger grids. The added code complexity and less flexibility of this functionality made it not worth using for this project.

It is possible that the initial loading of shared memory will introduce some initial overhead. Also, a barrier is required to ensure that all the data have been moved by the GPU threads.

4.4 Benchmarks

A simple procedural scene is used to benchmark the ray tracer. The scene grid has the size of $64x32x64bricks = 512x256x512voxels$. The benchmark scene consists of terrain generated with the methods discussed in Section 3.6. The scene also consists of a "doom guy" model. The voxel model is loaded through the application's custom vox format loader, which is discussed in Section 3.8.6. All voxels of the "doom guy" model are assigned the metal material type to increase scene material complexity since the terrain includes lambertian/diffuse and dielectric materials. Figure 4.4 illustrates the scene that we benchmark.

We collect benchmark data using the module discussed in Section 3.7. Three key data points are included in the produced benchmark report: the minimum, maximum, and average frame time. This data can tell a lot about system performance while remaining simple. The minimum and maximum frame times will show outliers in the benchmark performance. The maximum frame time is compelling because it can have a noticeable effect on the end-user. If the average frame time is optimal but the maximum frame time is too high, we know there was a visual lag on the screen at some point.

The code for our benchmarking and the application is available to the public and can be accessed on GitHub. We also include a branch in the repository named "benchmark" that keeps the mentioned scene configuration[21].

4.4.1 Results

Table 4.2 show the results of our benchmarking. The benchmark was performed on five different GPUs. Each GPU had a unique machine with CPU and other hardware. There was no extra effort in performing the benchmark on the same machine with different GPUs because, in every profiling session performed on all machines, the GPU was the bottleneck for performance, except for the RX 6800 XT. The hardware is listed in Table 4.1. The internal resolution of the application was set to $width = 1024, height = 576$ and then up-sampled bilinearly to native resolution in all our tests.

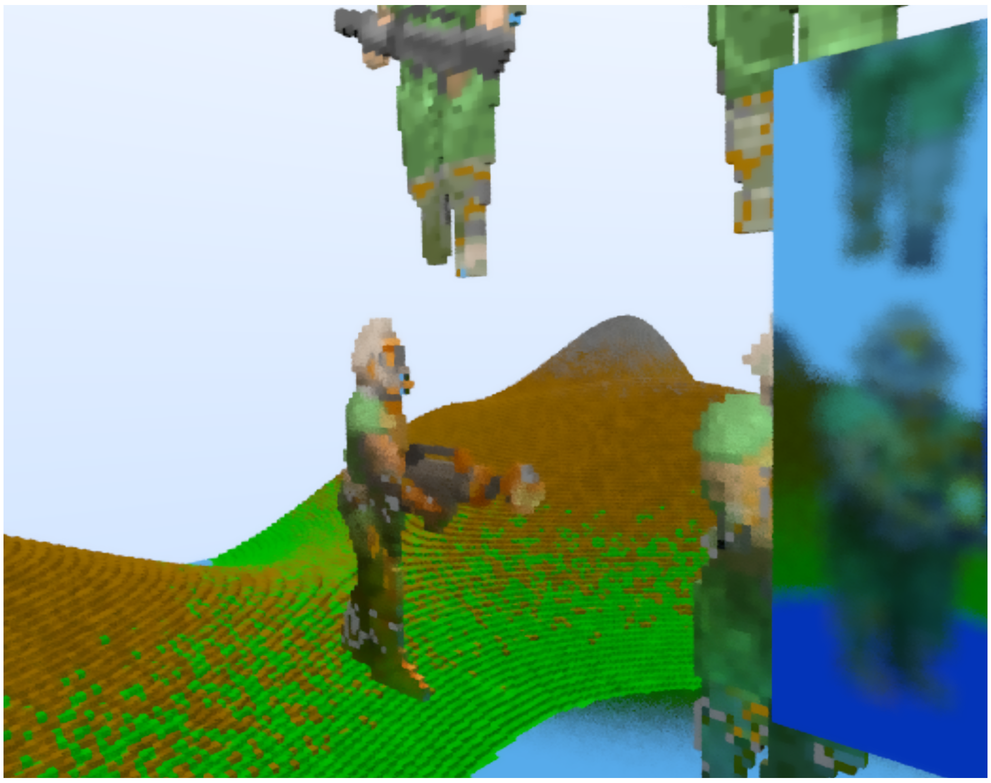


Figure 4.3: The loaded doom guy model viewed through dielectric water voxel in benchmark scene

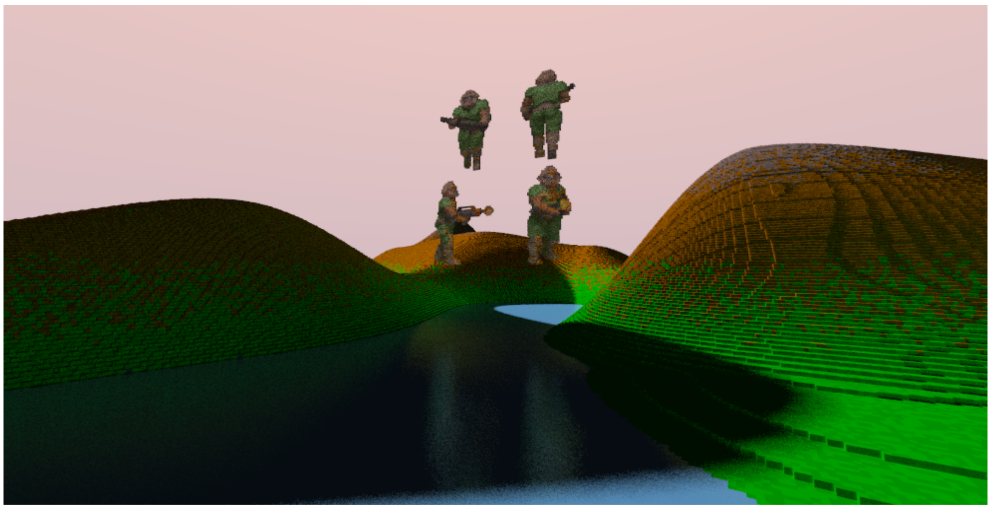


Figure 4.4: Wider image of benchmark scene

The data shows acceptable performance on all hardware given that application settings are adjusted according to GPU performance. In order to be able to claim

Table 4.1: Hardware used for each machine

GPU	CPU	RAM	PCIe	OS
GTX 1650 M	Intel i7 9750H	16GB 2666MHz	3.0	Ubuntu 20
GTX 1080 Ti	Intel i7 6700k	16GB 2133MHz	3.0	Windows 10
RTX Titan	AMD R7 5800X	32GB 3200MHz	4.0	Ubuntu 20
RTX 3090	AMD R7 5800X	32GB 2666MHz	3.0	Ubuntu 20
RX 6800 XT	Intel i7 6700k	16GB 2133MHz	3.0	Ubuntu 20

Table 4.2: Benchmark result table. All results are measured in milliseconds (lower is better)

Complexity 1: Max bounce 1, Sample count 1, Sun disabled			
GPU	Min frame time	Max frame time	Avg frame time
GTX 1650 M	1	40	13.573
GTX 1080 Ti	3	9	5.317
RTX Titan	0	16	2.190
RTX 3090	0	9	1.724
RX 6800 XT	1	5	1.983
Complexity 2: Max bounce 1, Sample count 1, Sun enabled			
GPU	Min frame time	Max frame time	Avg frame time
GTX 1650 M	1	46	18.087
GTX 1080 Ti	4	14	7.086
RTX Titan	1	12	2.754
RTX 3090	0	9	2.142
RX 6800 XT	1	6	2.107
Complexity 3: Max bounce 3, Sample count 2, Sun enabled			
GPU	Min frame time	Max frame time	Avg frame time
GTX 1650 M	57	237	108.166
GTX 1080 Ti	22	99	43
RTX Titan	7	30	13.753
RTX 3090	5	25	10.459
RX 6800 XT	2	11	5.789

real-time performance, the frame times should be less than 33*ms*, which would mean a Frames Per Second (FPS) greater than 30. The benchmark is performed with three different presets *Complexity 1* which is a max ray bounce of 1 with only one ray per pixel. The sun is also disabled in this preset. *Complexity 2* is the same as *complexity 1* except the sun is enabled. At *complexity 3* max ray bounce is set to 3, and 2 primary rays are sampled per pixel. The sun is also enabled in this mode.

The first samples are from a laptop with a GTX 1650 M. The 1650 machine performs worse than all the tested units, which are directly related to the GPU being the weakest in the test set. A high max frame time can be observed already at the simplest setting. High frame times might be related to thermal throttling since laptops have worse cooling than desktops. The average frame times is acceptable at $13.573ms = 73fps$. The GTX 1650 M fails to achieve real-time performance When the ray tracing requirements increase.

The second GPU tested is the GTX 1080 Ti. 1080 performs substantially better than the 1650 M. The average frame is $< 8ms =< 125fps$ both with and without the sun enabled. We get an average of $43ms = 23fps$ on the most expensive settings tested, which is below a playable frame rate. A middle ground between *complexity 2* and *complexity 3* would probably result in more acceptable performance while maintaining acceptable visual fidelity.

The RTX Titan and RTX 3090 achieve acceptable frame rates on all levels. We see at the lowest complexities that the minimum frame time goes as low as 0ms. Zero frame time means the measurement was less precise than the timer could record. The average frame time of the RTX Titan at *complexity 3* is at $13.753ms = 72fps$, and with a max frame time of $30ms$, we see the card performing at its limits.

Finally, the most recent card in the test bed is the AMD RX 6800 XT. On all complexity levels, we see good performance. Even at the highest complexity, the worst frame time observed is $11ms = 91fps$. The source of the large discrepancy in performance between AMD and Nvidia in the benchmark is unknown. As in Section 3.8.5, memory read is a major bottleneck for the ray tracing step. AMD advertise a feature of their hardware called *infinity cache*. AMD state:

> *With up to 128MB of ground-breaking AMD Infinity Cache acting as a massive bandwidth amplifier, get up to 3.25x the effective bandwidth of 256-bit 16 Gbps GDDR6 memory*[22]

There might be some truth to this statement. The RX 6800 XT uses GDDR6 memory, while the RTX 3090 uses GDDR6X. GDDR6X has a memory bandwidth of $21Gbs$ while the GDDR6 has a bandwidth of $14Gbs$. RX 6800 XT's almost double performance might be the improved caching through the infinity cache on the AMD hardware since in the VRAM of the RX should be slower. The boost clock of the two cards is also substantially different and might affect the performance difference. The RX 6800 XT has a boost clock of $2155MHz$ while the RTX 3900 has a boost clock of $1725MHz$.

4.5 Why benchmark these GPUs?

Overfitting is when optimization is applied for one class of hardware, and performance suffers on untested hardware. Tests are performed on different GPUs based on vendor and generation to make sure that our results are not based on overfitting. GPUs have considerable variation in features that might affect performance. Hotspots typical for all hardware should be identified and prioritized.

Chapter 5

Conclusion & Future Work

In this chapter we summarize the findings of this thesis. This includes our research goals, methodology, and our main results. A longer discussion of aspects of the project that can be improved is also covered as there are many interesting ways to extend this work.

5.1 Conclusion

This research had two goals: to deliver real-time voxel ray tracing on non ray tracing hardware and to extend the brickmap data structure to support material data. This paper presented a method for storing color and material information that can be used both for offline path tracing and real-time ray tracing. A Vulkan application was also made to utilize this method. The process of building the Vulkan application is documented and presented in this thesis to some degree. Tools we used throughout development have been included and described to help others who wish to develop similar applications. Design of stable data streaming from CPU to GPU with the Vulkan API have also been covered.

The presented *material buckets* is a system that delivers fast allocation and freeing of brick material data while remaining trivial to implement and use. The result of the bucket allocations are agnostic to both CPU and GPU, so the memory it controls can be on either or both. We also describe using this system in conjunction with multiple threads in a lock-less manner.

Combining the *material buckets* with the previous brickmap enables path trace or high fidelity ray tracing. A reference implementation of the brickmap data structure with the new material data was also presented. The project goals were very high, so there are more tasks to do before we can claim real-time ray tracing on GPUs without dedicated ray tracing hardware.

The slowest card tested was the GTX 1650M, which showed an average frame rate of $74fps$ while testing the basic settings. However, the card failed to scale to the most complex benchmark, which had an average frame time of $108ms$ or a frame rate of $9.26fps$. As discussed in Section sec: profiling, significant performance gains have been observed from optimizing memory read when profiling. Further memory and miscellaneous optimizations should increase the performance to the degree that even the GTX 1650M can do more complex ray tracing. More powerful cards like the GTX 1080Ti show promising performance on all complexities tested, but optimizations are still required. AMD RX 6800 XT shows exceptional frame rates on all tested complexities and is well within the real-time performance range. Future work is required to achieve stable real-time performance for the oldest and weakest hardware tested, but the data shown in the benchmarks indicate that it is a viable goal. The newest generation of GPUs shows good performance even though ray tracing hardware is not utilized. These GPUs have higher boost clock, faster memory, and more compute units than their predecessors.

5.2 Future work

The priority of future work should be optimizing the Vulkan code. Profiling has made it clear that optimal data layout and reducing data usage are two critical features to achieving good ray tracing performance. The presented method of storing material and color information does not utilize compression. Alternative modifications of the material storage could achieve even better cache coherency to improve the final frame time while offering the same material features. The code for the project is open-source, and anyone can contribute to the codebase[23]

5.2.1 Denoising and upscaling

Many production-ready ray tracing applications offer denoising and upscaling features to reduce the overhead of any ray tracing step. Nvidia has a proprietary Software Development Kit (SDK) called DLSS. Some have praised DLSS for being competent in denoising and upscaling system. However, there are some significant downsides to DLSS. It utilizes Nvidia's tensor cores, which only work on the Turing and later Nvidia architectures. Integrating DLSS in an open-source project is challenging since it is a proprietary SDK.

AMD offers two open-source alternatives called FSR 1.0 and FSR 2.0. Both FSR versions are designed to work on more than just AMD hardware, meaning it works on all modern GPUs and many older ones. FSR 2.0 has been compared favorably with recent versions of DLSS in terms of visual quality[24]. FSR 2.0 is not released to the public yet, but integrating FSR 1.0 and 2.0 when it is released could help improve the quality of the final image.

5.2.2 Ray buffers to improve work balance

Ray tracing in the application is performed in one single ray tracing step. Each GPU thread is assigned the same pixel to perform ray tracing on each frame. Render time is compounded between render frames by repeatedly assigning the same ray tracing job to the same GPU thread. A ray buffer system can remove some of the compounded time overhead. For each frame, a compute shader will dispatch primary rays that are put in a buffer. At the same time as the primary ray buffer is filled, another compute shader will grab incoming rays from the buffer and calculate ray color. The ray buffer eliminates the relation between a pixel and a given thread. A ray buffer can also be further developed to split the mentioned scatter behaviors covered in Section 3.2. Three unique scatter/ray trace shaders (1 per material type) can then be dispatched in parallel to perform unique scatters. Since one scatter function exists per compute shader, unique scatter dispatches will remove any warp divergence introduced by the scatter functions. Warp divergence is discussed in Section 2.2.1.

5.2.3 Improved noise functions for ray tracing

The current ray tracing implementation utilize white noise. The system utilizes noise for two functions. The first function is ray direction modifiers for multiple ray samples per pixel. When the application is sampling more than one ray per pixel, subsequent rays must use a modified ray direction to get a better image result. The scatter functions are the second part of the ray tracing system that uses noise. All material scatter functions mentioned in Section 3.2 utilize white noise currently to scatter the ray. When comparing white noise sampling with blue noise sampling, the white noise is more apparent in the final output[25]. It is also harder for denoisers to denoise images produced with white noise because of the more apparent noise.

Replacing the current noise functions with blue noise sampling would reduce the need for multiple samples, thereby improving the visual fidelity and the performance of the raytracing shader.

5.2.4 High Dynamic Range Rendering

Traditionally in rasterization rendering, the output uses a color range of $[0,1]$ so that each color channel is limited to this range. The normalized color range is usually sufficient when the color is easy to control. In some rendering applications, this can cause images to become over/underexposed, and detail is lost. High Dynamic Range (HDR) solves this problem by increasing the color range for framebuffers. By increasing the color range for the target image, more detail can remain visible. A new exposure variable is introduced to decide how much color values should contribute. This process is called *tone mapping*[26].

Introducing HDR to the project's codebase would increase the visual fidelity. It would also be required if support for multiple lights were added since color blending of multiple emitting lights is likely to cause over-exposure.

5.2.5 ReSTIR GI: Path Resampling for Real-Time Path Tracing

ReSTIR GI: Path Resampling for Real-Time Path Tracing is a recent publication that presents an effective path sampling algorithm[27]. Integrating the resampling presented in this paper would increase the application's performance and visual fidelity.

5.2.6 Improve Normal Calculation

Currently normals are calculated using a somewhat wasteful procedure:

```
// calculate the voxel position
voxel_position = (position * scale);
// calculate the center coordinate for the voxel
const vec3 center = voxel_position + vec3(scale * 0.5);
vec3 normal = hit.point - center;
const vec3 abs_normal = abs(normal);
// find the axis with the highest value and use only this axis as the normal.
// this is valid since the voxels are axis aligned
normal.x = sign(normal.x) * float(abs_normal.x >= abs_normal.y && abs_normal.x >=
    abs_normal.z);
normal.y = sign(normal.y) * float(abs_normal.y >= abs_normal.x && abs_normal.y >=
    abs_normal.z);
normal.z = sign(normal.z) * float(abs_normal.z >= abs_normal.x && abs_normal.z >=
    abs_normal.y);
hit.normal = normalize(normal);
```

Code listing 5.1: Wasteful normal caluculation

Normal values are implicitly known if a step has been performed in the voxel grid traversal. If the previous step for getting to a voxel was $prev\_step = (-1, 0, 0)$, then we know that the current hit normal is $normal = -prev\_step = (1, 0, 0)$. The voxel step is not currently used because it does not account for any initial hits before any steps are made. Combining the intersection test algorithm discussed in Section 2.1.2, we can compute the normal while checking if the initial ray hits the voxel grid. We can then start by initializing a normal and then use the previous step if any step has been performed.

5.2.7 Continuous Integration using Github

Unit tests exist in the codebase as discussed in Section 3.8.1. These tests help check for regression in the codebase, but running tests is not enforced when committing. Continuous Integration (CI) helps with this issue by running tests on incoming commits and pull requests. `github.com` which is the current host of the project source code, offers easy support for CI using their *workflow* system[28]. The project should utilize Github's workflow functionality to avoid accepting changes that break current unit tests.

5.2.8 Hashing of buckets and deduplication

The execution time of the ray tracing shader is heavily dependent on memory retrieve, as we discuss in Section 3.8.5. Buckets can be hashed by inspecting the material slice that the bucket has ownership over. If two or more buckets have an identical hash, we can consider the two buckets as one and make both bricks use the same bucket. By reusing buckets, we reduce the material data used and subsequently increase the render performance by avoiding cache misses. A downside of the mentioned segmentation of global memory in Section 3.4.1 is that this deduplication can not be trivially done between worker threads.

Bibliography

[1] T. van Wingerden and J. Bikker, 'Real-time ray tracing and editing of large voxel scenes,' M.S. thesis, Department of Information and Computing Sciences, Jun. 2015, p. 32. [Online]. Available: `https://studenttheses.uu.nl/handle/20.500.12932/20460`.

[2] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005, ch. 37, ISBN: 0321335597.

[3] T. L. AB. [Online]. Available: `https://teardowngame.com/`.

[4] H. Nguyen, *Gpu Gems 3*, First. Addison-Wesley Professional, 2007, ch. 21, ISBN: 9780321545428.

[5] M. Flynn, 'Very high-speed computing systems,' *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: `10.1109/PROC.1966.5273`.

[6] T. Lottes. [Online]. Available: `https://gpuopen.com/learn/vulkan-device-memory/`.

[7] F. Sans and R. Carmona, 'A comparison between gpu-based volume ray casting implementations: Fragment shader, compute shader, opencl, and cuda,' *CLEI Electron. J.*, vol. 20, 2017.

[8] N. Capodieci and R. Cavicchioli, 'Vkpolybench: A crossplatform vulkan compute port of the polybench/gpu benchmark suite,' *SoftwareX*, vol. 15, p. 100 793, 2021, ISSN: 2352-7110. DOI: `https://doi.org/10.1016/j.softx.2021.100793`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2352711021000996`.

[9] R. Cavicchioli, N. Capodieci, M. Solieri and M. Bertogna, 'Novel Methodologies for Predictable CPU-To-GPU Command Offloading,' in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, S. Quinton, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 133, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 22:1–22:22, ISBN: 978-3-95977-110-8. DOI: `10.4230/LIPIcs.ECRTS.2019.22`. [Online]. Available: `http://drops.dagstuhl.de/opus/volltexte/2019/10759`.

[10] K. Enarsson, 'Particle simulation using asynchronous compute : A study of the hardware,' M.S. thesis, Blekinge Institute of Technology, Department of Computer Science, 2020, p. 46.

[11] Khronos, *Execution model*, Available at `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap3.html#fundamentals-execmodel`, 2021.

[12] P. Shirley, *Ray tracing in one weekend*, Dec. 2020. [Online]. Available: `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.

[13] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn and W. Jarosz, 'Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting,' *ACM Trans. Graph.*, vol. 39, no. 4, Jul. 2020, ISSN: 0730-0301. DOI: `10.1145/3386569.3392481`. [Online]. Available: `https://research.nvidia.com/sites/default/files/pubs/2020-07_Spatiotemporal-reservoir-resampling/ReSTIR.pdf`.

[14] G. contributers. [Online]. Available: `https://www.glfw.org/`.

[15] B. Taudul. [Online]. Available: `https://github.com/wolfpld/tracy`.

[16] B. Karlsson. [Online]. Available: `https://renderdoc.org/`.

[17] AMD. [Online]. Available: `https://gpuopen.com/rga/`.

[18] AMD. [Online]. Available: `https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf`.

[19] C. Hebert and C. Kubisch. [Online]. Available: `https://developer.nvidia.com/vulkan-memory-management`.

[20] ephtracy. [Online]. Available: `https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt`.

[21] A. Hjerpbakk. [Online]. Available: `https://github.com/Avokadoen/zig_vulkan/tree/benchmark`.

[22] A. M. Devices. [Online]. Available: `https://www.amd.com/en/partner/radeon-rx-6000-series`.

[23] A. Hjerpbakk. [Online]. Available: `https://github.com/Avokadoen/zig_vulkan`.

[24] A. Klotz, *Amd fsr 2.0 goes head-to-head with dlss and works on 'all' gpus*, T. Hardware, Ed., May 2022. [Online]. Available: `https://www.tomshardware.com/news/amd-fsr2-deathloop-vs-dlss`.

[25] A. Wolfe, N. Morrical, T. Akenine-Moller and R. Ramamoorthi, *Rendering in real time with spatiotemporal blue noise textures, part 1*, Dec. 2021. [Online]. Available: `https://developer.nvidia.com/blog/rendering-in-real-time-with-spatiotemporal-blue-noise-textures-part-1/`.

[26] J. de Vries, *Hdr*. [Online]. Available: `https://learnopengl.com/Advanced-Lighting/HDR`.

[27] Y. Ouyang, S. Liu, M. Kettunen, M. Pharr and J. Pantaleoni, 'Restir gi: Path resampling for real-time path tracing,' *Computer Graphics Forum*, vol. 40, no. 8, pp. 17–29, 2021. DOI: https://doi.org/10.1111/cgf.14378. eprint: https://d1qx31qr3h6wln.cloudfront.net/publications/ReSTIR\%20GI.pdf. [Online]. Available: https://research.nvidia.com/publication/2021-06_restir-gi-path-resampling-real-time-path-tracing.

[28] G. docs contributors, *About continuous integration*. [Online]. Available: https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration.