Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Improvement of voxel-based rendering for haptic-oriented virtual dissection

Master's Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science (M.Sc) in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

Submitted by

B.Sc. Felix Riehm

Submission Date:

Berlin, 07.04.2022

First Supervisor: Prof. Dr.-Ing. Johann Habakuk Israel

Second Supervisor: M.Sc. Lucas Siqueira Rodrigues

Abstract

Dissection and preparation of a fossil can be a cumbersome task for paleontologists. To assist researcher with these tasks, virtual reality combined with haptic feedback offers new ways to segment and explore. To achieve that, an existing prototype provides the functionality to cut material from a computed tomography scan of a fossil. This thesis improves this prototype by implementing a custom voxel library, increasing the performance, and solving five usability problems. The resulting implementation can render and modify about 2.4 millions visible voxels in real-time at 90 fps or above. This is achieved with Unity DOTS, GPU instancing, parallel computing and a native chunk-based data structure. Furthermore, it also introduces filtering, undoing, restoring, more precision when interacting with the volume using physic colliders, collision with custom meshes and a new way of initialising the cutting process. The frequency the haptic feedback renders improves from 50 hz to 90 hz.

Kurzbeschreibung

Das Sezieren und Präparieren eines Fossils kann für Paläontologen eine mühsame Aufgabe sein. Um Forscher bei diesen Aufgaben zu unterstützen, bietet die virtuelle Realität in Kombination mit haptischem Feedback neue Möglichkeiten zur Segmentierung und Erkundung. Um dies zu erreichen, bietet ein bestehender Prototyp die Möglichkeit, Material aus einem Computertomographie-Scan eines Fossils zu schneiden. In dieser Arbeit wird dieser Prototyp durch die Implementierung einer individuellen Voxel-Softwarebibliothek verbessert, die die Leistung des Prototyps erhöht, und zusätzlich fünf Probleme der Benutzerfreundlichkeit löst. Die resultierende Implementierung kann etwa 2,4 Millionen sichtbare Voxel in Echtzeit mit 90 fps oder mehr darstellen und verändern. Erreicht wird dies durch Unity DOTS, GPU-Instanzierung, paralleles Rechnen und eine native Chunk-basierte Datenstruktur. Darüber hinaus werden Filterung, Rückgängigmachung und Wiederherstellung von Voxeln, mehr Präzision bei der physikalischen Kollision mit dem Volumen, Kollision mit individuellen Polygonnetzen und eine neue Art der Initialisierung des Schneidprozesses eingeführt. Die Frequenz, mit der das haptische Feedback aktualisiert wird, verbessert sich von 50 hz auf 90 hz.

Contents

1. Introduction

The introduction chapter states the motivation, the goal and the approach of this thesis.

1.1. Motivation

This project is part of the "Virtual Dissection" project which is a branch of the research program "Matters of Activity". "Virtual Dissection" is concerned with "the potential utility of virtual reality technologies for anatomical and paleontological dissections and preparations" [1]. To explore this potential, data from a computed tomography scan are used in a virtual environment where a user can interact with the scanned object by using adequate tools. The virtual environment which serves as a prototype is based on Unity, and the data of the object scan is represented as a voxel grid. Currently, only a basic drill and a magnifying glass exist in the application. The drill uses haptic feedback with the Touch [2] device from 3D Systems. "Virtual Dissection" aims to provide researchers with a new way of working by assisting them with algorithms and spatial exploration of data.

With the existing prototype, several usability problems emerged from a heuristic evaluation. These problems address haptic feedback, rendering and tools. Currently, the haptic feedback of the drill feels too coarse and the spheric tool head doesn't allow for precise work. In addition to that, there are concerns about the performance of the prototype when manipulating the voxel grid.

1.2. Goal

The goal of this thesis is to improve the rendering of the current prototype by focusing on solving five usability problems which emerged from a heuristic evaluation. The usability problems being the desire to restore voxels, to filter unwanted density data, to undo changes done to the fossil, to add more precision while cutting and to change the way cutting is initialised. The solution to the usability problems must be implemented, and tested with a heuristic evaluation and performance tests. Furthermore, the emphesis is on improving the current prototype. Thus, the technology stack is not intended to be swapped with an entirely new one. This thesis therefore focuses on using a game engine and an external voxel library to solve the problems. These constraints form the research question: "How can the voxel grid rendering of the existing prototype be improved by focusing on the five mentioned usability problems, using a game engine and a voxel library?".

1.3. Outline and approach

In chapter "3. Specification analysis", the usability problems of the heuristic evaluation of the existing prototype are categorised related to "haptic feedback", "tools" and "rendering". The usability problems this thesis tries to solve can be found in that categorization. As part of the voxel grid rendering improvement, in chapter "4. Performance analysis of the existing prototype", the prototype is evaluated in regard to the performance when cutting a volume. This gives information about why the software library is not sophisticated enough and therefore is insufficient. In chapter "5. Voxel library comparison" candidates for a replacement of the library used in the existing prototype are compared. The comparison includes performance tests and several metrics described by the utilised methodology further explained in section "5.1. Methodology". Despite the comparison a custom voxel library has been implemented which can be found in chapter "6. Implementation". To verify the performance improvement of the implementation, in section "7.1. Benchmark", performance tests are shown and compared to the existing library. The newly added functionalities are verified by an heuristic evaluation in section "7.2. Heuristic evaluation". And finally in "8. Future improvements" future improvements of the implementation are discussed.

2. Fundamentals

In this chapter, the basics of this thesis will be described. The first section will describe what virtual palaeontology is and how it refers to virtual dissection. The second section will focus on the data representation which is used in the existing prototype and in the implementation, and the third section will describe what the setup of the existing prototype looks like. In the last section, related work will be discussed.

2.1. Virtual palaeontology

Virtual palaeontology is concerned with exploring fossils visualised with three-dimensional data. The data usually come from tomographic or surface based methods. The benefit of having a 3D visualisation of a fossil is that it contains morphological information, which is hard to extract the traditional way where one uses physical or chemical methods inflicting damage to the fossil. Even with 2D visualised methods dissecting a fossil with a series of image slices using tools like 3D Slicer [3] [4] and searching for these morphological data can be a cumbersome work where virtual palaeontology might be helpful. According to Sutton et al. studying fossils with 2D data may not fall into the category of virtual palaeontology [5, p. 1]. Having these morphological data in 3D space allows for easier segmentation where one removes material of the fossil to make the interior structure of the fossil more visible. Since the fossil is digitalized, one can also color parts and edit it without damaging [5, pp. 1-2].

There are many ways of obtaining 3D data from a fossil. Sutton et al. [5, p. 6] create a taxonomy dividing them into tomographic (e.g. a CT scan which provides image slices of the interior of the fossil representing the densities of the fossil) and surface based methods (e.g. laser scanning producing a point cloud on the surface of the fossil. Tomographic methods can further be divided into destructive and non-destructive. Rahman and Smith [6] provide an alternative overview of 3D imaging techniques, including information about scan resolution and suitability.

To display tomographic 3D data, it has to be registered, which means that all image slices have to be aligned to another. CT scans do this naturally, while others, like physical-optical scans, do not. One can extract isosurfaces or vector surfaces out from it. Isosurfaces are points in a 3D space sharing the same intensity which make up a surface while vector surfaces use splines to define the object [5, pp. 13-14]. To improve this reconstruction process, one usually prepares the data virtually by enhancing density contrast to cope with

CT scan noise which can blur the edges of the fossil [7, p. 4], and by filtering unwanted values which map to air, bedrock or parts from the rock matrix which are not of interest. After preparing, one usually does dissection by applying segmentation, which in case of virtual dissection involves image processing techniques but applied in 3D space. One usually first determines surfaces by identifying and mapping density values within a range to an object (e.g. a type of bone). After that, one can let those regions grow to a desired size which better segments the object. However, this is often not sufficient. One also has to mask parts of the fossil by adding or removing values from a region. This is a cumbersome task which usually is done slice by slice in a program like 3D Slicer [7, pp. 5-7].

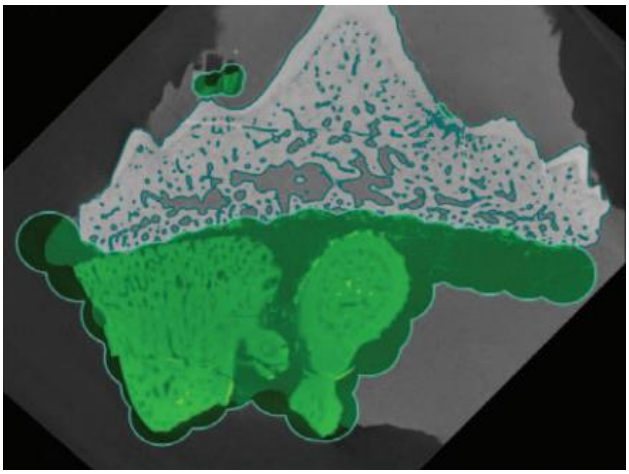

Figure 1: Masking selection (green) of a tomogram of a fossil. One must do this for possible hundreds of continuous slices [7].

The visualization is done by rendering polygons, a volume, or a point cloud [5, p. 15]. In Figure 2, one can see usual reconstructs flows where the flow used in the existing prototype of this project is coloured yellow.

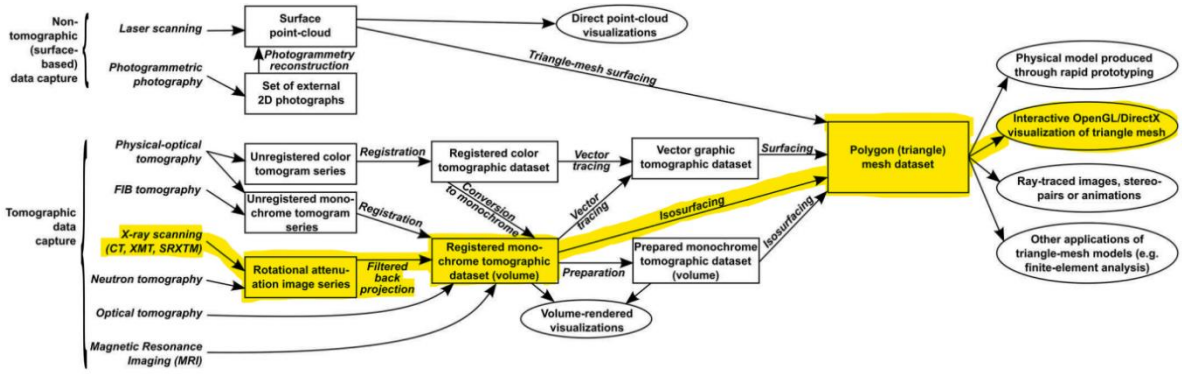

Figure 2: Reconstruction flows of fossil data. The yellow marked path is used in the existing prototype of this project. [5]

2.2. Voxels

A volume element (also known as "voxel") is an element inside a 3D grid or, more broadly speaking, inside a volume. Usually, it represents a cubic space at its position in the grid. The position is determined by its relation to the other voxels in form of an index. Different information can be mapped to a voxel: a value which determines if the voxel is outside of a represented object or not, colour, density, and other values. Voxel grids have a wide variety of applications. The most significant feature of a voxel representation of an object is that you can manipulate or visualise the inner volume. Therefore, voxel grids are popular in video games where the player can shape the procedural generated world by adding or subtracting voxels, and in medical, scientific or film domain where one wants to visualise different layers of an object or animate a volume like water or clouds [8, p. 578].

The memory needed to store a voxel grid can be high considering the O(n^3) behaviour of a cubic voxel grid. This can be optimised however, the neighbourhood relations of the voxels allow for the usage of an octree [8, pp. 824-827] (see Figure 3) to represent the grid which is also effective for searching inside the grid (e.g., when collision detection is computed). The relation of the voxels also helps when doing occlusion culling [9]. This is done by voxelisation of the scene geometry, linking neighbouring voxels and building a graph from which one can determine at runtime if an object should be occluded. For instance, Unity uses Umbra for its occlusion culling, though at least with Unity, culling only works for objects not created at runtime [10]. Thus, if voxel grids are generated at runtime, one has to write its own occlusion culling.

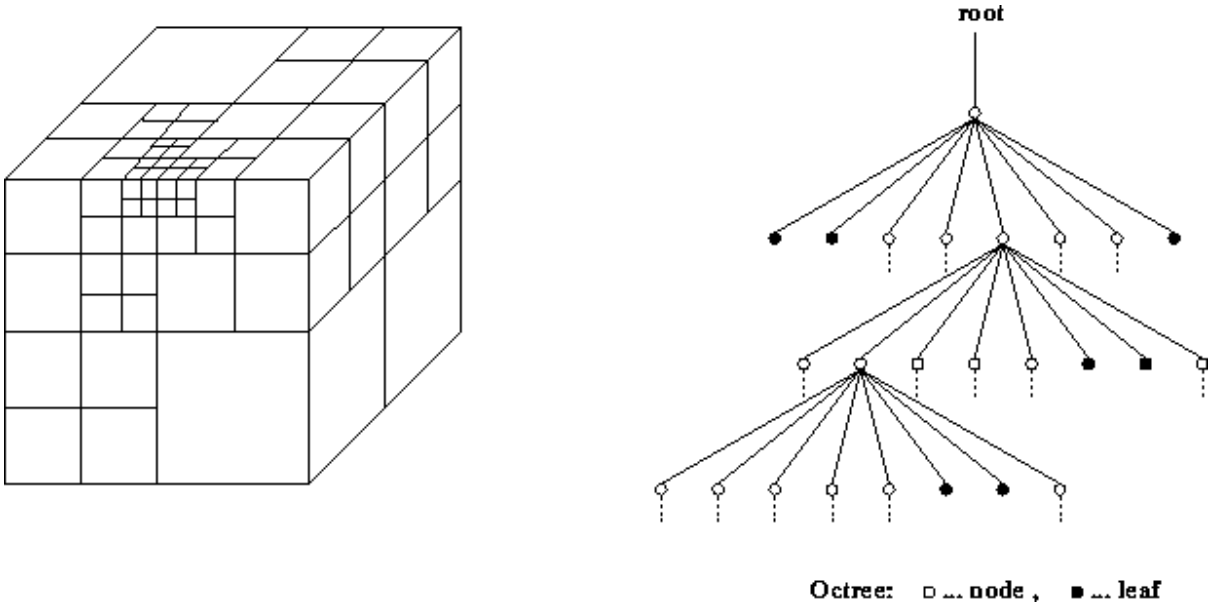

Figure 3: Example of an octree. Every node is divided by 8 sub nodes. It works similar to a quadtree which is commonly used to divide 2D spaces while octrees are usually used for dividing 3D spaces [11].

The octree representation can be further optimised with a sparse voxel octree [12] where unnecessary voxels (similar or empty) are discarded. Generating a voxel grid is called

voxelisation and can be done with an existing mesh [13], point cloud or medical stacked images slices from a CT scan (see Figure 4) [8, pp. 578-579]. Usually, for medical CT scans, voxel grid sizes are smaller than for archaeology because a living being can only take a limited amount of radiation. The actual size of a voxel grid for a CT scan, which can take up several to hundreds of gigabytes, depends on the number of slices, the size of the object and the voxel resolution [14, pp. 441-443].

Rendering a voxel grid can be done in different ways and differs from rendering solid or translucent objects. For solid objects, there are methods based on ray tracing which makes interaction with the object more difficult but allow for a better looking rendering result, and others, which convert the voxels into a polygon mesh by either representing each voxel as a polygon cube or by generating a smoother mesh with the marching cube algorithm [15] [8, pp. 582-583]. For translucent objects, one can use volumetric ray tracing where rays go through an object and scatter. The existing prototype of this project interprets the fossil as a solid object and uses polygons to represent voxels. A comprehensive look at translucent rendering can be found in the work of Akenine-Moller et al. [8, pp. 589-649].

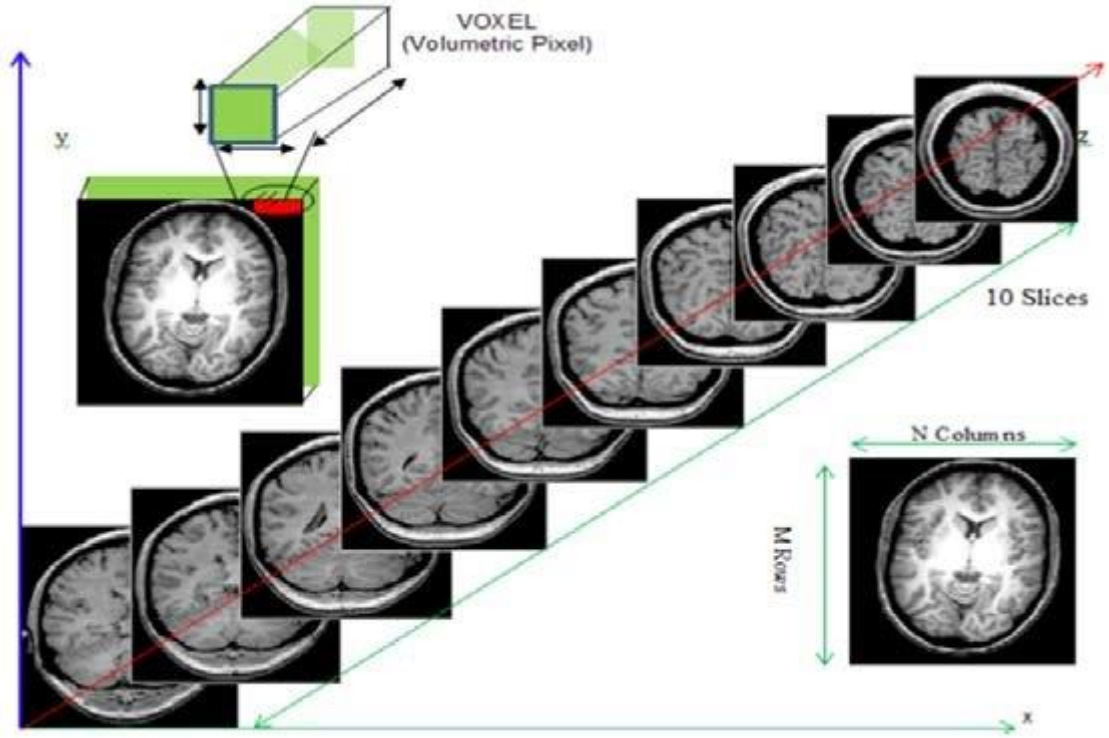

Figure 4: Voxelisation of stacked image slices of a brain MRI scan [16]. For each slice a value of an area of the 2D image is mapped to a voxel on a corresponding voxel grid slice.

2.3. Existing prototype

There is an existing prototype for the "Virtual Dissection" project described in "1.1. Motivation" and will be later evaluated in "4. Performance analysis of the existing prototype". The prototype is a project for Unity 2019.4.12 and uses SteamVR [17], OpenHaptics [18] and Cubiquity [19] (see Figure 5).

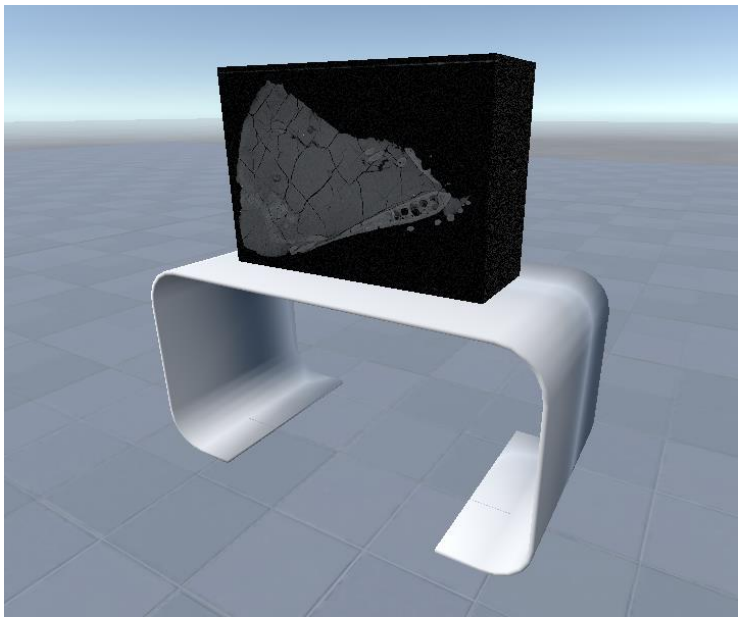

Figure 5: Unity scene of the existing prototype.

SteamVR is responsible for making the Unity scene compatible with VR headsets (in this case a Valve Index) and OpenHaptics from 3D Systems provides the functionality to operate the Touch device which is used to create haptic feedback when exploring the fossil (see Figure 6).

Figure 6: Devices used for the existing prototype. Left: Touch device [2]. Right: Valve Index [20].

Cubiquity is a voxel library and is used to represent the fossil data in a volumetric form. Cubiquity is introduced later in "4.1. Cubiquity". The fossil data comes from a transversal CT scan in form of a series of images. Each image comprises grayscale values ranging from 0 to 255, each representing the density of the fossil. The prototype renders only a part of the CT data. The 784 original image slices have a resolution of 3010x2048 and therefore could make

up a voxel grid of 3010x2048x784. The prototype downscales the resolution to 512x348 and only uses 176 slices located in the middle of the original scan. This makes it a 512x348x176 voxel volume. The down scaling improves performance and high resolution is often not necessary when studying a fossil, however, varies case by case. Mallison [21, pp. 33-34] discusses resizing of CT scans and gives examples.

2.4. Related works

In chapter "5. Voxel library comparison" a comparison of voxel libraries has been made. There are a lot of methodologies and frameworks for comparing software. For instance, Boloix et Robillard [24] describes one which is mostly suited for bigger projects where one wants to incorporate producers and managers on a project level, system administrators and software engineers on a system level and end users on an environment level. Cohill et al. [23] suggest a compact methodology which focuses on functionality, usability, performance, support and documentation, and will be used later for the comparison in this thesis. It produces a score which can be easily compared to other software evaluated this way. Furthermore, a user can also apply weighting to the categories and metrics used in those categories. Gediga et al. [25] give more context about the goals and results of an evaluation. For instance, the evaluation used for the comparison in this thesis is a summative evaluation which asks "Which one is better?". It will deliver a decision which library is best fitting for improving the prototype regarding the concerned usability problems. It does not tell how the libraries can be improved. Brown et Wallnau [26] mention product-oriented and process-oriented technology evaluation where process-oriented evaluation would mean to look for another approach of dealing with the rendering like a point cloud based approach, and product-oriented evaluation would mean to search a library which replaces the current one with a better one with similar approach. The comparison of this thesis can be classified as product-oriented analysis. Brown et Wallnau also mention a genealogy and habitat model that helps to understand the relationships between software, technology and specification. This is done by creating a graph with links and nodes. Furthermore, the work of Jadhav et Sonar [25] of gives a comprehensive list of software selection methodologies and evaluation techniques and criteria. In their literature review, they found several evaluation techniques: AHP (analytical hierarchy process), WAS (weighted average sum), feature analysis, and a fuzzy based approach. The first two are the most common, while the second one is also used in this voxel library comparison. There is also an ISO standard 9126:1991 [28] for software evaluation criteria for software evaluated as a product by the view of a user.

Most works regarding the overarching goal of the "Virtual Dissection" project are focused on medical applications. And most of them are just displaying tomography scans in a 3D scene or tagging it with "immersive" by putting the 3D scene into virtual reality to use it for surgery planning or learning. Furthermore, the interactive parts of those works usually just provide functionalities to change views, or rotate and transform objects in the 3D scene [22] [23] [24] [25]. This is not true for the existing prototype since it involves cutting the object directly on voxel level in conjecture with a haptic device. The work of Rizzi [26], for example, does not fall into this category. His work is concerned with immersive surgical simulation by using *ImmersiveTouch* [27]. He found a haptic rendering technique based on voxels rather on polygons. Furthermore, he combines these two by using voxels primarily for haptics and renders the object with the marching cube algorithm, which results in a polygon mesh. He is

also presenting ways of drilling and cutting materials with his approach. Doing this also involved regenerating a mesh locally when modifying the voxel grid. He achieved this in real-time by using Nvidia CUDA and occupying a GPU thread per voxel to generate the triangles. Reddivari and Smith [28] show another work which allows for cutting. They developed a VR visualisation tool for MRI images in Unity where a user can interact with the object with tools. They decided to build their own voxel engine, which supports physical interaction and allows for cutting. For input, they used a leap motion device to track hand motions. A work from Faludi et al. [23] allows no cutting, but supports haptic feedback. They show an approach to explore tomographic data in virtual reality using haptic feedback. They developed a haptic rendering algorithm which provides continuous force based on voxels. And for visualisation, they used direct volume rendering using *SpectoVR* [29] which specialises in displaying medical data in VR with advanced ray tracing techniques. They found that Unity, which they used for visualisation, was not reliable for haptic calculation and switched to *CHAI3d* [30] which guaranteed better response time. It was not possible to modify objects by cutting into it in their application. It was possible to draw guidelines on objects. The evaluation yielded positive results regarding line accuracy when using haptic feedback versus not using it. Zhang et al. [31] also incorporated haptic feedback into their surgery simulator by using Unity, Nvidia Flex and two Touch devices for each hand.

There are also works who try to combine Unity with 3D Slicer. For example, King et al. [24] created a Unity application which shows a user multiple CT scans, loaded with 3D Slicers, in virtual reality. The communication between 3D Slicer and Unity is done by a web server. Another interesting approach using Unity demonstrated Wheeler et al. [25] by taking advantage of OpenGL context sharing. They managed to display objects loaded and visualised with VTK (a C++ visualisation toolkit) in a Unity scene. The benefit being that VTK (also used by other tools like 3D Slicer) is aimed at medical imaging and has support for displaying CT scans as volumes. They later did a clinical evaluation about it, which resulted in a strong positive result [32].

Other works [33] [34] [35] implement direct volume rendering, which is used widely in medical visualization tools. A little more distant related work from Escobar-Castillejos et al. [36] describes an architecture to build a visuo-haptic application with a game engine. Like in the existing prototype of this project, they used Unity to make their case. Haptic-oriented simulations are often written in C++ and use raw OpenGL for graphic output. With their written Unity plugin, they propose an architecture consisting of two modules which handle the connectivity of haptic devices and the information exchange to Unity.

3. Specification analysis

Prior to this thesis, a heuristic evaluation of the existing prototype with three specialists was conducted. For the evaluation, the heuristics of Sutcliffe and Gault [37] were used, which are based on Nielsen's work [38]. There are also other works who try to evaluate a virtual environment, like the work of Murtza et al. [39] or Gabbard et al. [40]. To extract the relevant usability problems for this thesis, the feedback was sumerised and put into three categories ("haptic device", "rendering" and "tools"). As they target different topics, each require their own focused work:

Compatibility with the user's task and domain
Haptic device: There are multiple problems with the force feedback. It was suggested that it could be improved by providing force feedback on edges of an area. A user should notice it when moving the head of an erosion tool accidentally outside of the desired area. In general, the force feedback was too harsh for a task which requires precision. Another suggestion was to use gradients when computing the force feedback instead of values solely based on grayscale values of single voxels.

Rendering: Besides the force feedback, there were also problems with the way a user interacts with the voxel grid. It was suggested that the application should have the ability to filter density values with a threshold to make carving and segmentation easier by removing unwanted material. Alternatively or additionally to that, selecting a value range as a mask to work on would also be beneficial in avoiding cutting unwanted values. It would be also a good feature to scale the volume, which allows to zoom in and thus providing better precision when cutting. Another request was to pan, scale and rotate the target object for a better experience, instead of having the object in a static vertical position. Because of the stationary haptic device, it would be also nice to put the object further away, so you don't have issues when cutting in a deep object or when wanting to get an overview of the object. Another crucial feature which was requested was the ability to undo changes made to the object either by undoing the entire last operation or by bringing back the material with the tool head of a separate undo tool.

Tools: For tool improvements, it was suggested that the spherical tool head for collision detection should be adjustable or replaceable with other, more precise ones. Adjusting the tool head properties (like the size or amount it subtracts materials), if implemented, should

not require shifting focus on a distant UI. Rather it would be convenient to manipulate such properties over an addition physical device like a foot pedal or a device which is hold in the non-dominant hand.

Natural engagement

Haptic device: The force feedback device did not match with the virtual counterpart. This is because the physical movement in the prototype was amplified to be able to carve on the full range of the virtual object, which was larger than the physical device movement range.

Rendering: It was found that the rendering of the CT object was unrealistic because it also included a lot of black empty voxels. These could be filtered, which would represent the target object better before starting segmentation.

Natural expression of movement

Haptic device: The haptic device has a limited range and can not be moved beyond its limits, which isn't represented in the virtual environment. This could be mitigated by allowing the user to move the target object.

Close coordination of action and representation

Haptic device: When users started carving sometimes there were a delay in haptic feedback when a voxel was already removed.

Realistic feedback

Haptic device: The force feedback was not appropriate when carving. It did not feel like they were touching materials. The force feedback has to reflect that better. A suggestion was to have continuous force when hitting the material and when approaching the point of material erosion.

Support for learning

Tools: The tool head deals continuous damage to the surface of the object, which is no pleasant experience if a user wants to rest his hands and put down the device. It was suggested to hold down a button on the device to activate the carving functionality of the tool.

There are five usability problems which this thesis will try to solve:

- The restoring functionality for removed voxels applied to a tool.
- The undo functionality when cutting or restoring.
- The filter functionality to delete voxels in a value range.
- Making the tool head more precise by allowing different shapes. This also includes making cutting in general more precise by using colliders.
- A better way of initialising cutting

4. Performance analysis of the existing prototype

Before deciding about any changes to the voxel grid library, the library of the existing prototype has to be evaluated to show what causes problems and why. For that, a benchmark (see 4.2 Benchmark) of the prototype is has been performed.

4.1. Cubiquity

The library of the existing prototype is called Cubiquity and was written in spare time by David Williams, a post-doctorate researcher who has been specialized in computer graphics [41]. Cubiquity originally released in 2013 and is the predecessor of PolyVox – a voxel library which was also developed by the same person and released 2006. Cubiquity is open source, provides classes to build and manage a voxel data set primarily for games, and is written in C++ (see Figure 7).

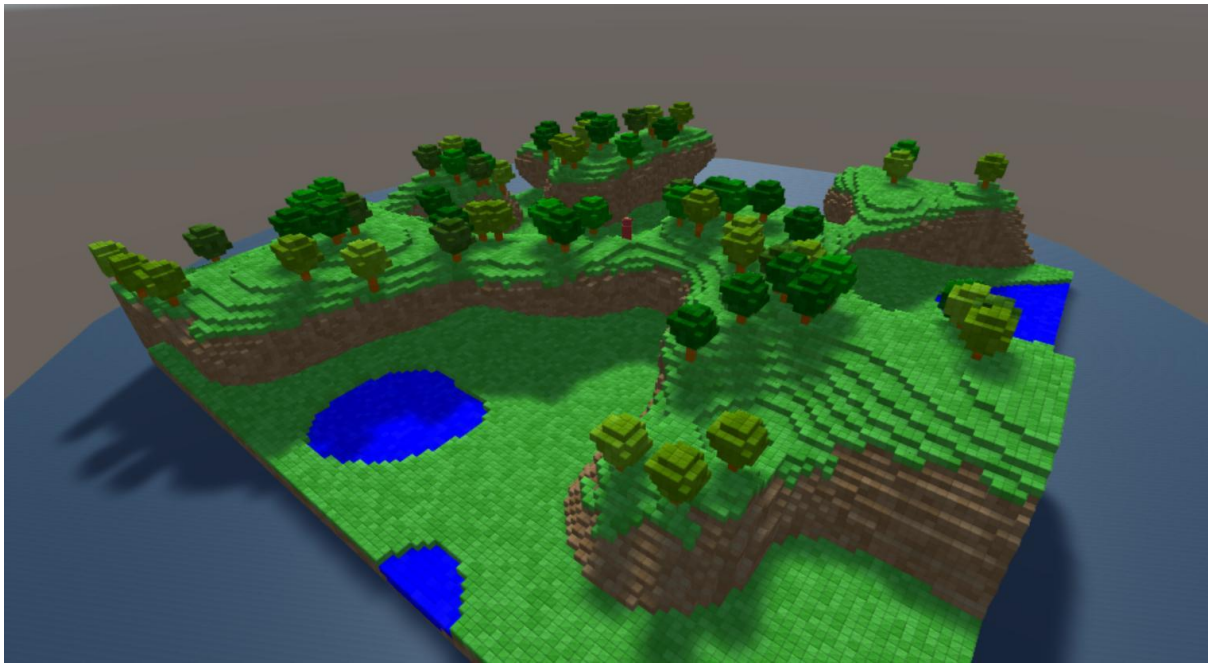

Figure 7: Sample scene of Cubiquity built with image slices [42].

However, the development and support halted in 2017 because of the development of the upcoming Cubiquity 2 [43] [44]. The big advantage of Cubiquity is that is written in C++

which makes it usable for different languages and game engines. It also provides a tool to generate a voxel database from images slices with the following command:

```
ProcessVDB.exe -import -imageslices /path/to/image/folder -coloredcubes output.vdb
```

The images need to have the PNG file format with an alpha channel and have to be numbered at the beginning of the file name with leading zeros. If you don't have the image slices as PNG files (like in my case) you can convert JPEG files to PNG files easily with ImageMagick [45]:

```
mogrify -format PNG32 -channel RGBA *.jpg
```

The database file which gets generated is called `.vdb` which stands for "virtual database" and is generated and used by SQLite [46]. This can be confusing because there is a popular voxel data structure which is also called `.vdb` (short hand for the keywords volumetric, dynamic (grid) and B+ (tree)) developed by Ken Museth [47]. It is used in voxel libraries like OpenVDB and NVIDIA GVDB Voxels, and can be used in 3D modelling software as Blender and Houdini.

For easier usage, Willimas created a Unity plugin [48] [49] which allows calling the C++ functions from Unity. This is done by a C interface of the Cubiquity C++ library (compiled as a .dll file) and by a C# wrapper file using the P/invoke technology [50] which passes C++ function calls through the .NET framework. The plugin also supports Unity editor tools and functions to create and manipulate a voxel grid. This Unity plugin is used by the prototype and will be evaluated by performance tests in the next section.

4.2. Benchmark

The benchmark of the existing prototype will be described by four sub-sections: "Setup", "Implementation", "Result" and "Analysis". The goal of the benchmark is to evaluate the performance problems the prototype may have when drilling, and to show why these happen.

4.2.1. Setup

For the testing, a Unity scene was created mirroring the prototype state of the Unity scene "Box Collider with Rigid Body" which was used for the previous heuristic evaluation. The scene consists of the same fossil data as in the prototype, a `ColoredVoxelVolume` with a size of 512x176x348 voxels (see Figure 8) and a scale of 0.003. Other than `TerrainVoxelVolume` which supports a set of materials for each voxel, a `ColoredVoxelVolume` is just made up of color values. Furthermore, the color values are quantized (see the class `QuanitzedColor`). This will reduce the precision of the color representation but will lead to more colors sharing the same value. This, on the other hand, means that the voxel grid can be better compressed on the hard drive. The head of the drill was also copied from the prototype with a default scale of 0.025. A UI was created as a starting point to execute benchmarks, to undo changes for restarting benchmarks, and to show the results of the benchmarks.

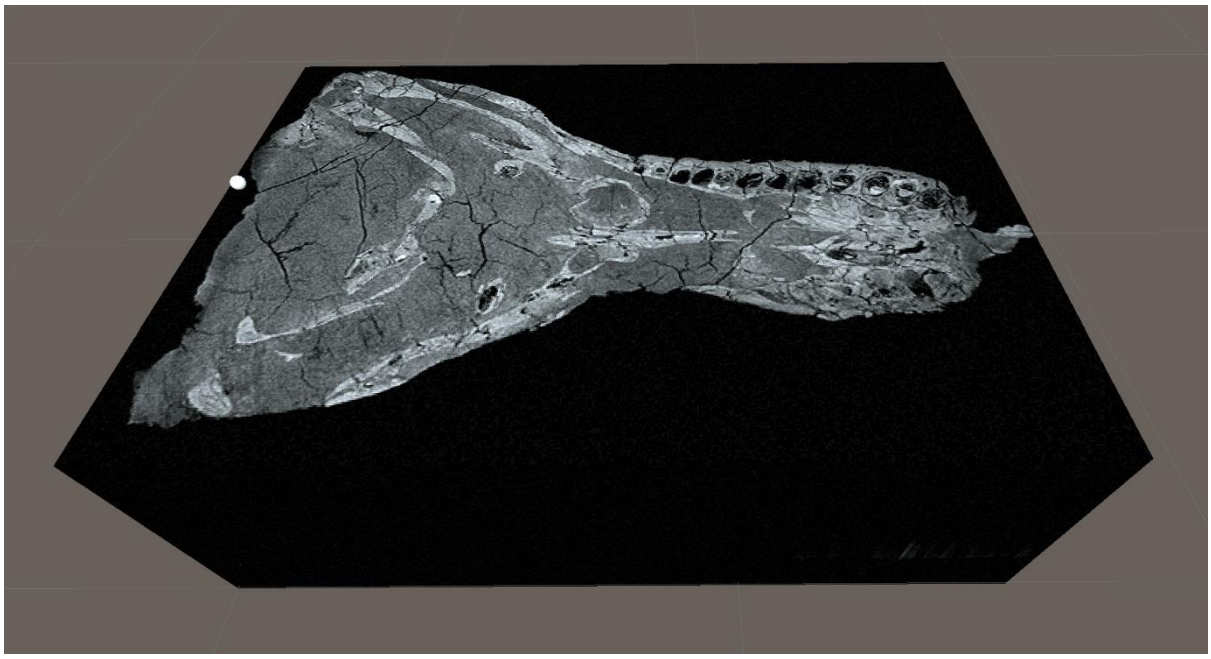

Figure 8: Unity scene used for the two benchmarks. Beside the fossil data, a small sphere can be seen which acts as the drill head.

To create comparable results between benchmark runs the haptic device was not involved. Instead, the drill head was moved inside the voxel grid via script to allow for collision detection to occur. The measurements were taken by the Unity profiler of a development build since profiling inside of the editor causes overhead and does not represent the actual performance of the application. Two benchmarks were created, one with different numbers of

voxel removed per collision hit and one with variant physic time step, both to see how the application scales.

4.2.2. Implementation

One aspect of the implementation is the collision handling of the drill head. The prototype does collision detection with ray casting, which is triggered by a `BoxCollider` that embodies the voxel grid. When the drill head is inside the `BoxCollider`, an event is triggered which handles the collision. The event handler casts a ray from the center position of the drill head in the direction the haptic pen of the phantom device is pointing. The ray will check for voxels with `Picking.PickFirstSolidVoxel()`. If the ray hits a voxel, that voxel serves as a center point for a sphere in which voxels are destroyed. The ray length was shortened for this benchmark to get comparable results. Why this had to be made and why this method of collision detection has a downside to it, is discussed in the section "4.3.1. Possible improvements". The way the voxels in the sphere are selected is calculated in a discrete way and is given by the code seen in Figure 9.

```
List<Vector3i> voxelsToDelete = new List<Vector3i>();
int rangeSquared = range * range;
for(int z = zPos - range; z < zPos + range; z++)
{
    for(int y = yPos - range; y < yPos + range; y++)
    {
        for(int x = xPos - range; x < xPos + range; x++)
        {
            int xDistance = x - xPos;
            int yDistance = y - yPos;
            int zDistance = z - zPos;

            int distSquared = xDistance * xDistance +
                              yDistance * yDistance +
                              zDistance * zDistance;

            if(distSquared < rangeSquared)
            {
                voxelsToDelete.Add( item: new Vector3i(x, y, z));
            }
        }
    }
}
```

Figure 9: Code for removing a sphere in the existing prototype.

`zPos`, `yPos` and `zPos` are the components of the voxel position. The `range` value represents the radius of the drill head and varies per benchmark. As already mentioned, benchmark one deals with different number of voxels removed per collision hit. The first run uses a range of 2, which removes in total 27 voxels. The second run uses a range 5 which will remove 485 voxels and the last run uses a range of 10, which removes a total of 4,139 voxels per collision hit.

Benchmark two is concerned with the time step of the physic system of Unity. The physic system is responsible for calculating collisions and triggers collision events on `MonoBehaviour`. It also does not run in sync with the main update cycle of Unity. Unity allows users to update their data based on the physic time step with the method `FixedUpdate()` while `Update()` is the regular method to change data and is bound to the frame rate. The physic system uses a fixed time step and is executed in the same main thread as the `Update()` method. If it can't catch up (e.g. because the last frame took longer than the fixed time step) it will be called multiple times in a frame. This separation of frame rate and physic calculation is important. If it would not be independent, it would mean that on a slower computer (which may produce fewer frames per seconds) physic objects would move slower, while with higher frame rates suddenly everything moves at a fast speed. Regarding the benchmarks, this means that if you increase the time step, more voxels will be deleted (if the drill head has moved enough distance in that time frame to reach new voxels). As already mentioned, `FixedUpdate()` can also be called multiples times during a frame which leads to multiple collision events that can lead to unwanted problems (like the removal of more voxels than intended, which then accumulates a delay between frames). The physic time step of the prototype was set to 0.2 seconds (it is Unity's default value and is ideal for a 50 fps experience). Why this is bad for the application will be mentioned in "4.3.1. Possible improvements". To see how the prototype scales regarding the physic time step, benchmark two sets this value to 0.02 in the first run, $0.01\overline{3}$ (75 collision events per seconds) in the second run and $0.0\overline{1}$ seconds (90 events per seconds and ideal for a 90 fps experience, which matches the default refresh rate of the two displays of the Valve Index) in the third run.

Different runs of a benchmark should be comparable; therefore, the measurement must not depend on the frame rate nor on the fixed physic updates. Only the time a frame usually takes to calculate is of interest (and of course, the parts it is made of). Because of that, the head drill movement is based on wall clock time. The timing of the physic system and the movement time are always set to the same value which aims to avoid `FixedUpdate()` to be called multiple times within a regular frame rate bound `Update()` cycle. Ideally, this triggers the physic system at the same time as the drill head moves. The position update is done inside the `Update()` method. If an update takes too long because the handling of the collision event took too long, the head is moved whenever the next update is called.

In both benchmarks, the drill head moves along a pre-defined path. This is important because the voxel library may handle the removal of voxels differently based on the octree

structure at the given voxel positions. For example, a set of voxels marked to be deleted could share more than one octree region (node with 8 children), because of that the tree has to be traversed differently. Or maybe some voxels in a region are already deleted, which may make searching for a voxel faster depending on the implementation. If the benchmark could cover most of the cases, it would be beneficial for the analysis. Because of that, the drill head moves in form of a conical helix through the volume. It follows the path of an equiangular spiral and is submerged inside the volume at all time (as seen in Figure 10). With this path, the drill head also moves farther into world space at the beginning of the path and then transitions into smaller steps. The end of the spiral reflects the actual cutting experience probably the most, since a user will move the drill head slowly along a surface.

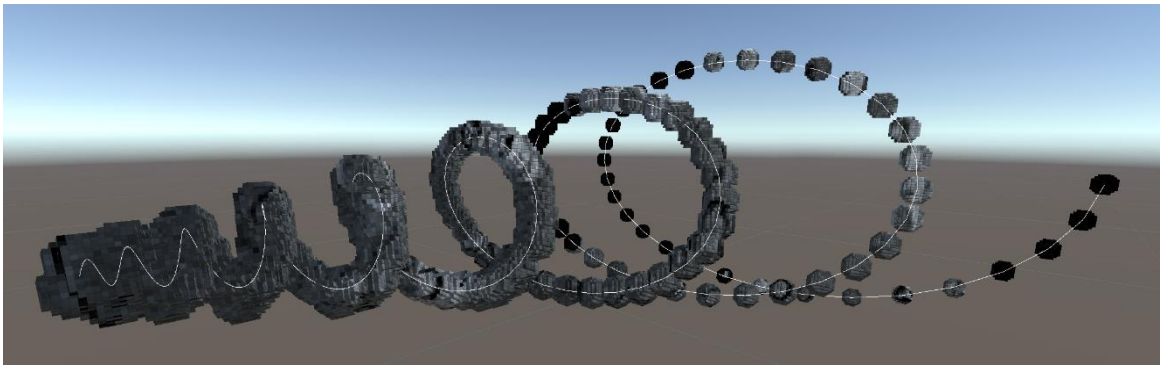

Figure 10: Conical helix with an equiangular spiral [51]. The surrounding volume cannot be seen. It covers most of the volume and has a part with sparse voxel removal which slowly transitions into a part with dense voxel removal.

The definition of the path is taken from mathematische-basteleien.de [51]. The step size is set to 0.1, the start value to 0 and the end value to 8π. This will make the drill move 251 times. The benchmark time is therefore determined by the size of the step the drill head moves, and by the frame rate, if a frame takes longer than the target time step between head drill movements. A high frame rate doesn't make the benchmark time go faster.

4.2.3. Result

For the two benchmarks, the Unity profiler listening to the development build of the project was used for measurements. The haptic device and the Valve Index were not connected to the system. The system used for the benchmarks has the following components:

- Intel Core i5-10400 with six cores, each having 2.9 to 4.3 GHz
- 16 GB DDR4-2666 RAM
- GeForce RTX 3060 with 12GB GDDR6

For all benchmarks, the frame rate was limited to 90 fps, and the resolution was set to 1600x1440. The frame rate is matching the default refresh rate of the Valve Index and the resolution matches the resolution of one of the displays of the Valve Index.

The limitation of the frame rate also helped to minimize the number of idle frames between the collision events, which helped when viewing the profiler diagram (fewer frames in the diagram make it more compact). When viewing the Unity profiler, you can only visit a limited range of frames. All frames past that range are deleted.

Benchmark one uses variant sizes of the range parameter mentioned before. The first run uses a range of 2, the second 5 and the third 10. To make the results more comparable to the results in "7.1 Benchmark" the range values will be displayed in the form of their sphere scale equivalent. In other words: How the scale of a sphere, given the volume scale of 0.003, must be to enclose the voxels that are being removed with a given range value. This is given by `Math.Floor(SphereRadius/VolumeScale)`. Thus, the range value of 2 is associated with a sphere scale of 0.015, range value 5 with scale 0.3 and range value 10 with scale value 0.6.

At the end of the spiral, it will happen that to-be-deleted voxels may already have been deleted from the last collision event. For all runs of benchmark one, a physic time step of $0.01\bar{3}$ seconds was used. The results can be viewed in Figure 11.

The frame time graph of B1R1 and B1R2 are quite similar while B1R3 is the worst. Occasionally, there are spikes in the first two runs, but overall, the performance is stable. If the spikes happen though, they are in between the 16 ms and 33 ms mark. The spikes, like in all other runs, appear to be random. If a run with the same setting is executed multiple times, the spikes appear in different locations and to different extent. The behaviour of the spikes of the benchmarks were expected to be the same, giving that the same voxels that are deleted. It is assumed that this noise comes from thread scheduling, interrupting the process, and the profiler running in the background. B1R3 consistently produces frame times above 11 ms and is considered not stable enough. Noticeable is also the absence of user script calculations at the end of B1R2 and B1R3. When approaching the end of the spiral, the distances between the drill head positions get smaller. Because of that, the ray does not collide with any voxel, or even if it does, the script removes fewer solid voxels and thus the time for rebuilding the mesh decreases. This effect is even more visible in B1R3. There, the range value of the voxel removal process is larger, leaving more empty space. And as expected, the time a frame spends inside user scripts increase with the number of voxels deleted per valid collision.

Benchmark two is concerned with the variation of the physic time step. The physic time steps for the three runs are 0.02, $0.01\bar{3}$ and $0.0\bar{1}$ seconds while the range value of the voxel removal process is held at 5. The first thing which can be observed is that when comparing B2R1 and B2R3, the benchmark times decrease. That is because the drill head moves faster along the spiral path when the physic time step gets smaller. This is, as already mentioned, because the time step between the head drill movement is set equal to the physic time step.

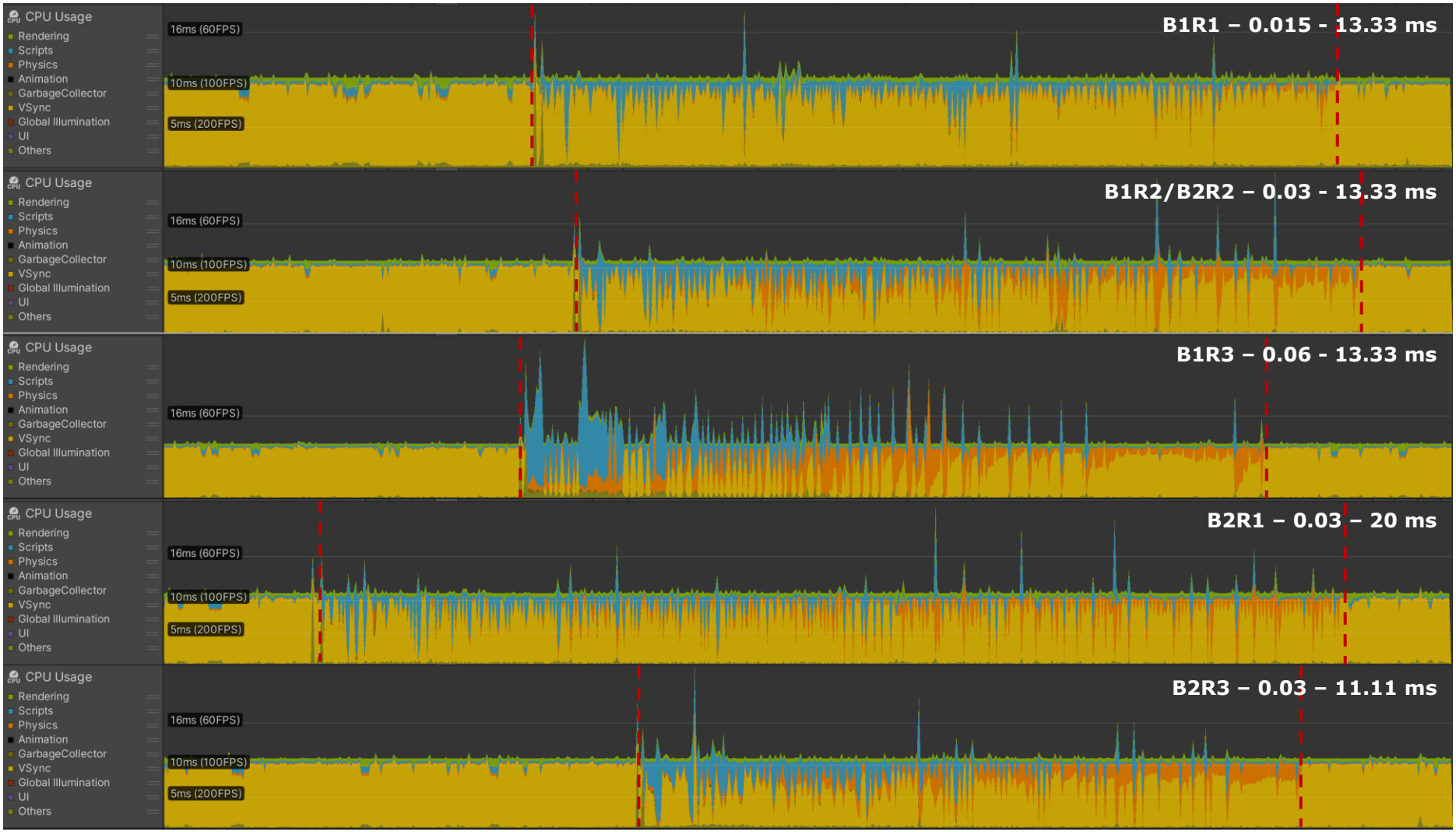

Figure 11: Performance when cutting the volume with different settings. Red lines mark the start and end of each benchmark. Label in the top right corner: Benchmark number and run number – drill head scale – physic time step.

As expected, there are also more collisions and less idle frames at the ending of B2R3 since there will be more collision hits in general with a faster physic time step. However, the performance with different physic time steps (B2R1, B2R2 and B2R3) appears to be the same. There are spikes in the diagram again, but overall, even with 90 physic updates per second, the performance is still stable. The times spend inside user scripts are roughly the same, since there is no change in the number of removed voxels.

4.2.4. Analysis

The benchmarks show that the prototype has no significant performance problem with the configuration it uses, the render budget of 11.11 ms is mostly met. However, the benchmarks show some spikes which are still bad for frame pacing and should be ironed out, even more if they appear constantly. Bad frame pacing may break the usability heuristic "Sense of presence" of the heuristics used in the previous heuristic evaluation. In addition to that, a user may want to make the sphere larger to remove more voxels at a time. It was shown that enlarging the sphere results in worse performance. To investigate further and find ways to increase the general performance, one of the worst frames was selected to investigate the parts it is made of. The chosen frame comes from the run with a physic time step of $0.01\bar{3}$ seconds and a range value of 5. For the investigation, deep profiling needs to be activated, which creates massive overhead. Because of that, the absolute numbers are not relevant, the relatives are more interesting at this point, assuming everything scales evenly. The hierarchy view of the profiler (see Figure 12) gives enough information for the investigation. The hierarchy view is an overview which shows the parts which make up the frame. It shows the times spent in these parts in percentage.

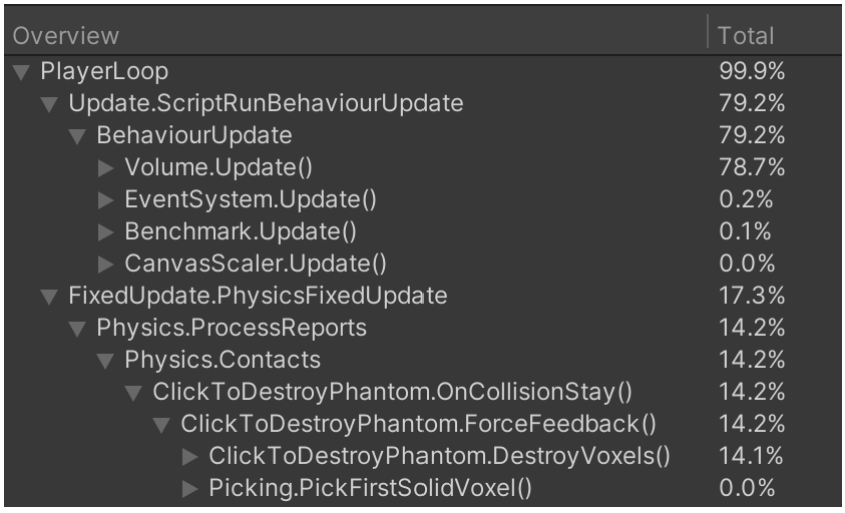

Figure 12: Extract of the hierarchy view of the deep profiler snapshot.

Unity's `Update()` function of different user scripts takes up the most time with 79.2%. The main cause of this is `Volume.Update()` which takes 78.7% of the total time. Another impact is the fixed update of the physic system which takes 17.3% of the total time. If we inspect

`Update.Volume()` further (see Figure 13), we see that `ColoredCubesVolume.SynchronizeOctree()` is the main cause.

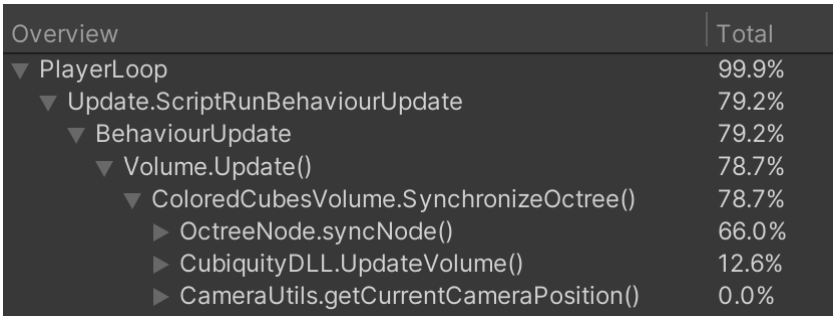

Figure 13: Same extract as in Figure 12 but with the "Volume.Update()" node expanded.

It calls `syncNode()` and `UpdateVolume()` which are the most taxing methods. Inside the physic update, the main cause is `DestroyVoxels()` which collects the voxels within a range and deletes them. 14% is caused by the methods from Cubquitity, first and foremost `SetVoxel()` from the `ColoredCubesVolume` class (can not be seen in Figure 12). `Volume.Update()` is dominated by `ColoredCubesVolume.SynchonizeOctree()` which syncs the mesh created by the C++ library with Unity's mesh data structure. Building the mesh is done by `CubiquityDLL.UpdateVolume()`. This method also incorporates the camera position to determine the LOD of the volume to discard voxels which are too far away. However, about 85% of `CubiquityDLL.UpdateVolume()` is caused by calling `OctreeNoce.SyncNode()`. This method is called recursively for every child a modified node has. These calls are dominated by the `MeshConversion.BuildMeshFromNodeHandleForColoredCubesVolume()` method. This method gets the mesh from the C++ library and builds the data for Unity's mesh data structure for the current node. For every node, it calls this method twice, once for the rendering-mesh and once for collision-mesh. The code comments mention that this is actually not necessary as building the mesh once would be enough, but for some reasons, they had some issues with it and left it building the mesh twice. However, the mesh for the collision is only built when the `ColoredCubesVolumeCollider` component is added to the volume in Unity. Even if this component is disabled, the mesh for the collider gets built. One has to remove the component to get rid of this functionality.

4.3. Discussion

The discussion section of the performance analysis of the existing prototype describes possible improvements you could make to the prototype to improve the performance and the user experience, without swapping the library. In the last sub-section it will be described how the functionalities and performance of Cubiquity relates to the usability problems aimed to be solved in this thesis.

4.3.1. Possible improvements

The main problem of the library is the data synchronization between Unity and the C++ library. Thus, the application is CPU limited, while the workload on GPU is fine. This relates to the mesh building inside the scripts of Cubiquity, the mirroring of the node data structure of OctreeNode.cs and OctreeNode.h, and maybe also the marshalling overhead created by lots of smaller calls like `GetVoxel()` and `SetVoxel()` of class `ColoredCubesVolumeData` (although not verified). That said, you would also have the problem with building the mesh when using Cubiquity with a purely C++ based game engine like Unreal Engine, since you would also have to sync the mesh to the Unreal engine mesh data structure. There are some possible improvements which can be done to Cubiquity. It is, however, questionable if these improvements should be done considering that the library is not under active development and support anymore. Furthermore, there are a lot of code blocks in the source code with comments which are questioning the effectiveness and purpose of their own code.

One quick and easy performance improvement would be to remove the `ColoredCubesVolumeCollider` component of the volume inside the prototype project. That removes the second build of the mesh. This is advised with the collision detection setup of the prototype anyway because the collision detection with the voxel grid is done with a ray cast instead of contact points coming from the sphere collider of the drill head. Having the ray cast method is fast, although it also leads to problems. There will be situations where the sphere mesh will hit surrounding voxels, but the ray cast won’t trigger because along the rays way there might be no voxels (see Figure 14). You would have to cast numerous rays to sample the surface of the sphere.

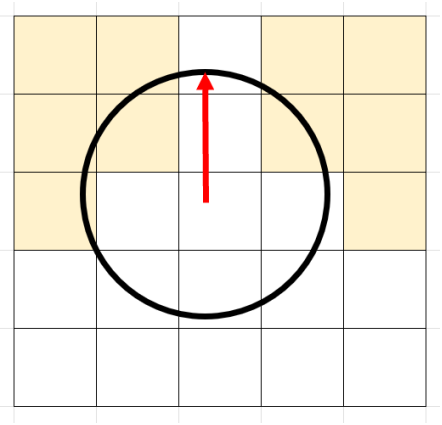

Figure 14: One of the problems when using a ray cast for collision detection: Some voxels are not detected. Black circle: Sphere mesh. Red arrow: Ray, which checks for voxels. White squares: Removed voxels. Yellow squares: Solid voxels.

Furthermore, if the ray cast is longer than the distance of the origin position to the sphere surface and the physic collision event triggers multiple times in a frame to catch up, then the ray may hit another voxel outside of the sphere radius causing more voxels to be deleted than desired in that frame. Using physic colliders would be a more precise but potentially more performance intensive alternative. If one wants to work with physic colliders and Cubiquity, one has to consider following: If the `ColoredCubesVolumeCollider` component is attached to the volume, the library creates colliders for each node, not for the volume bounding box. One has to manually write event handlers inside `OctreeNode.cs` to handle hit events from the physic system. This does not come out of the box from Cubiquity.

Another improvement which could be implemented is to utilise multi-threading. A potential candidate for this is Unity DOTS. However, to have thread safe access to methods, this must be supported by the library. For instance, there is the potential to execute the voxel removal code in parallel instead of executing `SetVoxel()` one after another. Maybe you can also convert the `OctreeNode.syncNode()` method to an iterative version from the recursive one which then could be made to be executed in parallel.

Furthermore, the user experience of the prototype could be improved by increasing the physic time step. When having the application running at a higher frame rate than the physic systems update rate, it can happen that the drill head moves through the fossil material without hitting it. The setup of the prototype uses a physic time step of 0.02 seconds. This problem can therefore occur and relates to the usability heuristic “Natural engagement”. However, when changing this value, one has to think about the haptic feedback which is connected to the collision event which then in turn happens more often, or even can trigger multiple times within a frame when the physic system tries to catch up because of performance issues. A statement from the heuristic evaluation addresses a mismatch between the haptic feedback and the visuals. This may correlate to that.

4.3.2. Usability problems

It is shown that the general performance when drilling is fine. But how does Cubiquity affect the usability problems aimed to be solved by this thesis?

One of these usability problems is to make cutting more precise by having different objects for the drill head. This requires colliders for collision detection and can be done with Cubiquity, but you would have to modify the source code to make it work. Inside `OctreeNode.cs`, you would have to write the collision handler by yourself to receive events when a different collider hits a voxel.

The next usability problem was a missing undo function. This does not come out of the box but can be implemented easily by storing the last deleted voxels in a list and then reapply them. The same goes for the voxel restoring function by a tool. Per default, it is not supported. In this case, however, it cannot even be implemented because there is no way with Cubiquity to ray cast against removed voxels.

The last usability problem was about filtering values. This does also not come out of the box but could be implemented by deleting the voxels which are within the value range. However, since Cubiquity is single threaded, this operation would take a long time, letting the user wait longer than necessary.

All four usability problems relate to adding and deleting a lot of voxels at the same time. As we can see from the benchmark B1R3 from Figure 11, this is a problem which can create a noticeable delay and let the user wait. Depending on the implementation of the tools for these functionalities, it might also break usability heuristic “Realistic feedback”. Therefore, a faster voxel library might improve this issue by reducing the delay or eliminate it. Thus, a comparison of possible voxel grid libraries is evaluated in the next chapter.

5. Voxel library comparison

In this chapter, Cubiquity will be compared to two other voxel libraries in order to find a better alternative. The methodology for this comparison will be described in the first section, after that the libraries that were chosen for comparison are described. A performance benchmark of all three libraries follows, to incorporate the data into the comparison. Finally, in the last two sections, the result and the conclusion of the comparison will be given.

5.1. Methodology

The comparison used in this thesis is based on a scaled-down version of the methodology of Cohill et al. [52]. The scaled-down version originated due the scope of this thesis which isn't focused on comparing libraries only. The methodology for evaluating application software of Cohill et al. addresses the quality of a software from a perspective of someone who wants to use it. The quality of software engineering, of how the software is built, is not in focus. It proposes five categories for an evaluation of a software: usability, functionality, performance, support and documentation. They are shortly explained in the next paragraphs.

Functionality

This tells you how much the software can do in terms of tasks it can support. A list of these tasks will be created for this purpose and will be later added to a master list. A score of 0 to 2 can be given to a task depending if the task is possible, hard to achieve or easy to achieve. In this comparison, this list is created only with the minimum capability set of functions the voxel library can provide. The reason for that is that the master list of all functionalities all libraries provide would be just too exhaustive. Hence, the score of 0 is never be given since the library would have been discarded in the first place later in the process, as it does not comply with the minimal capability list.

Usability

Usability refers to "how easy or difficult it is for the end user to access and implement functions provided by the package to perform application tasks." [52, p. 15]. Originally, the paper refers to 28 different criteria which can be grouped into 9 design objectives which in another turn relate to 3 software design goals ('easy to use', 'easy to lean' and 'flexible'). In this comparison, the design goals are evaluated in a summarized fashion without going through 28 different criteria since 28 criteria would be too exhaustive for this comparison, plus, the different criteria are not listed in full detail in the original paper, neither there is a

reference to it. It is only mentioned that it was developed by one of the authors. Therefore, the criteria for this comparison are just 'easy to use', 'easy to lean' and 'flexible'. They will be scored from 0 to 2 where 0 means not true, 1 partially the case and 2 fulfilling.

Performance

Benchmarks of the tasks list are utilised to compare the software. Since the benchmarks will be artificial, they do not reflect the actual implemented task performance, but can be used to compare the chosen software. For this comparison, the benchmark time values will be mapped from worst to best (0 to 2) by comparing it to the benchmark values of the other libraries. Performance tests will also just be made for four specific tasks on the minimal capability list to reduce the implementation afford.

Support

This category will tell you "the willingness and ability of software manufacturers and distributors to support users" [52, p. 16] in a rough manner. To review this, you can use the literature provided by the manufacturers or conduct interviews with them. The criteria are customer support, software control, training quality, and warranty provisions. Warranty provisions will be discarded since it is not applicable, software control will be discarded since there is no description of it. The scoring system is the same as in the "Functionality" category.

Documentation

This evaluates the quality of the documentation (manuals, how-to's, examples, etc.) offered by the manufacturer. Originally at this point, Cohill et al. refer to 'organization', 'typography and legibility', 'language', 'graphics and illustrations' and 'physical characteristics' [52, p. 16]. However, 'typography and legibility' and 'physical characteristics' are out-dated and not suitable for today's documentations, why only 'organization', 'graphics and illustrations' and 'language' are used. Additionally to that, the criterion 'Completeness' is added to give an estimation about the coverage of the code in terms of code documentations, examples, etc. In this comparison, the same scoring system as in the "Functionality" category is used for the criteria.

The methodology requires defining a minimum capability set of functions a software must provide for the user. This list is derived from the master list, which holds all functionalities all libraries provide. After that, each software will be evaluated according to the five categories. This results in scores for each category. At this point, one can add weighting to the scores depending on the focus of the comparison. The scores of each category come with different maximum values but are not related to each other. Because of that, they will be normalised between a range of 0 and 4 for better comparison. If needed, one can also add another weighting to the categories after the normalisation is done. Now all scores are inserted into a table for comparison. The table also includes a row for the average sum of each category to give a software a final single score.

The minimum capability set is defined by the following functions:

- Key functionalities (used for performance benchmarks)
    - Initialise a fixed sized voxel grid
    - Add voxels
    - Delete voxels
    - Modify voxels
- Needed for tool implementation
    - Ray cast against a voxel
    - Voxel collision with a custom mesh (e.g. drill head)
- Needed for rendering
    - Mesh generation
    - Shader to represent simple color (e.g. represent fossil densities as gray scale values)
- Needed for performance:
    - GPU instancing
    - Multi-threading
    - Grid represented with chunks, hash tables, trees or similar
    - Occlusion culling
    - Mesh optimization (e.g. greedy meshing)
- Others
    - Disk serialisation
    - Change of the voxel size

Other popular functionalities, like the marching cube algorithm to generate a smooth surface, LOD, nav mesh for AI pathfinding or networking, are not required.

5.2. Library selection

For the comparison, the libraries Cubiquity, Voxel Play 2 and OpenVDB 9.0.0 were chosen. Cubiquity is the library already in use in the existing prototype and which might be replaced. Voxel Play 2 was already purchased by a person associated with the project and is the best rated voxel library on the Unity asset store. OpenVDB is a high performance and award-winning voxel library used by DreamWorks, Pixar, Houdini and more.

It was also considered switching to the Unreal engine and use "Voxel Plugin" [53] which is another voxel library only available for the Unreal engine. Voxel Plugin is primarily targeting game development and is developed by Victor Careil. However, after a short evaluation, it was found that the library has a heavy focus on generating terrain with designer-centered tools. These tools can be used with the Unreal front-end or via code. This means, however, that a general purpose way of creating a voxel grid, or deleting a voxel, does not exist. For example, one has to use a sphere tool brush to create a voxel grid, even when using code to access the library. This doesn't fit the project and would also require implementing the VR and haptic device code in an Unreal project. Though, that might be not a huge problem since a SteamVR version for Unreal already exists. Most discouraging about this option is the switch to a different engine itself. Besides the different way Unreal works compared to Unity, Unreal's documentation and community forums are mostly focused for designer who primarily work with visual nodes to code their application. This is also true for Voxel Plugin. The documentation of Voxel Plugin for coding barely exists. Only a few examples and descriptions can be found. The focus is on the typical Unreal user who uses visual nodes. The experience when reaching out to their support was subjectively also not great. Another library which was considered was GVDB Voxels from Nvidea. However, this library only runs on Nvidea graphic cards, which is a downside for future hardware environments. However, if one wants, one could swap OpenVDB with GVDB Voxels in future since it is built upon OpenVDB and is essentially just a GPU accelerated version of OpenVDB.

In the following sections, the libraries are described with the exception of Cubiquity, which was introduced in "4.1 Cubiquity".

5.2.1. Voxel Play

Voxel Play 2 [54] is a paid Unity plugin from Kronnect for creating voxel-based worlds (see Figure 15). Kronnect creates assets for Unity and Unreal since 2015 and were nominated for the "Publisher Store Publisher of the Year" in the Unity Awards 2020 [55]. Voxel Play's primary use case is for games and thus comes with a lot of functionalities, which makes it fast and easy to build a game like Minecraft. Voxel Play 2 was released in August 2021 and is still supported by the development team. It has a rich documentation including examples and how-to's [56]. It has everything needed for purposes of this project: collision detection with meshes or ray casts, hidden surface removal with greedy meshing, utilisation of multi-threading, GPU instancing and octree-based occlusion culling for chunks.

Figure 15: A scene rendered with Voxel Play 2 [54].

5.2.2. OpenVDB

OpenVDB [57] is an open-source C++ voxel library targeting primarily the film industry and the scientific and medical field. It is developed by Ken Museth for DreamWorks Animation and is still under development and has active support. OpenVDB operates on the CPU and uses multi-threading, but there is also a GPU accelerated version of it from Nvidia, called Nvidia GVDB Voxels [58]. GVDB scales similarly to OpenVDB but is in general 10%-30% faster [59]. OpenVDB introduces its own file format .vdb which uses a custom compression technique based on topology and hierarchical data structure. It is designed for sparse volumetric data and therefore uses a SVO (sparse voxel octree: octree, which only stores surface voxels) like tree structure. However, OpenVDB uses a customized B+ tree with a fixed depth, which leads to many benefits. For example, when traversing or building an octree like Cubiquity uses, one has to traverse more nodes since an octree usually has a greater depth and also more

internal nodes [58]. OpenVDB saves nodes by compressing the voxels as a bit mask. The full data structure can be seen in Figure 16.

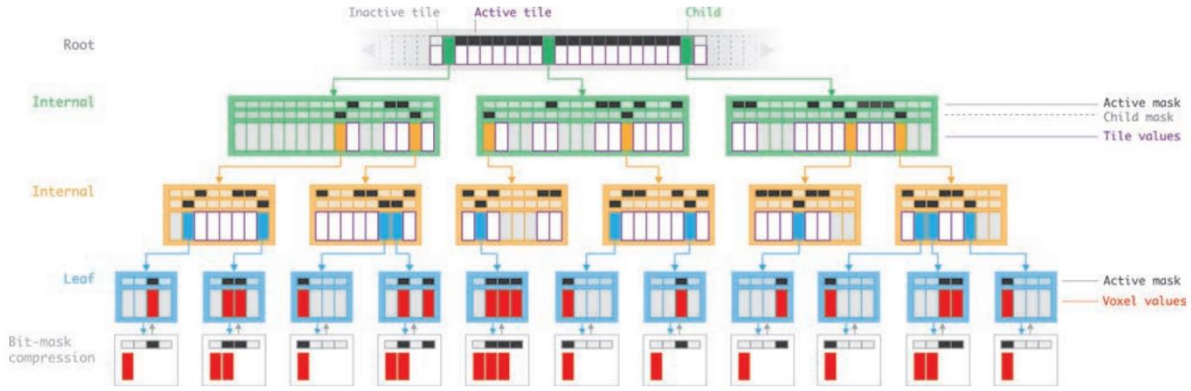

Figure 16: Data structure of OpenVDB [57].

The data structure is divided into four levels: The root, level three, stores internal nodes and is represented as a hash map. It can have any number of internal nodes and is only limited by the bit precision (e.g. 2.147.483.647 for a 32-bit signed integer). The topology therefore is dynamic and can be configured (having a height balanced octree thereby is possible). Level two consists of internal nodes which have, by default, a branching factor of 5 and therefore holds a grid of $2^5 * 2^5 * 2^5$ internal nodes which reside at level one. Level one also consists of internal nodes which hold, per default, a grid of $2^4 * 2^4 * 2^4$ leaf nodes. Leaf nodes at level zero store a bit-mask which stores, per default, a grid of $2^3 * 2^3 * 2^3$ voxel values. The branching factors are purposely set to the base 2 logarithms of the grid dimensions (e.g. a branching factor of $5 = \log_2 32$ will make level two have a grid dimension of $32 = 2^5$) as this allows for fast bit operations when searching the tree. How the data structure does look in action can be seen in Figure 17.

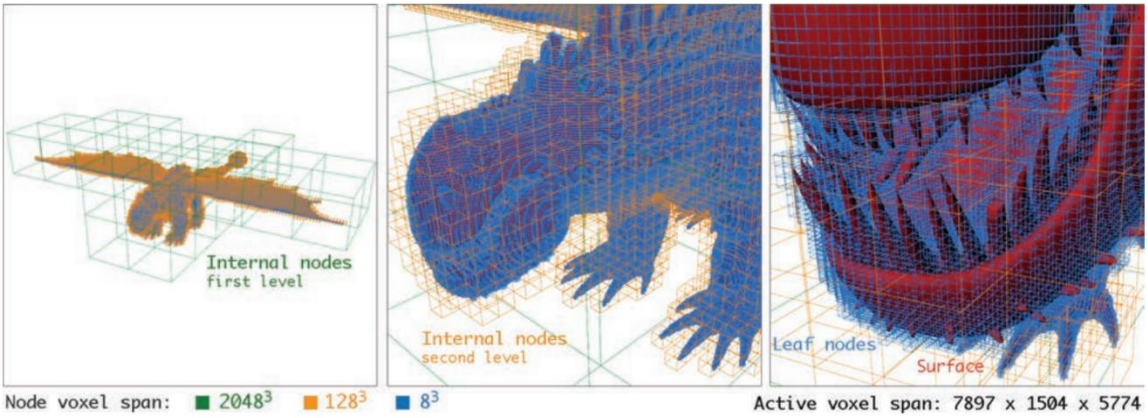

Figure 17: Rendering example of the OpenVDB data structure [57].

There is also the concept of tiles, which groups voxels to a node. Then there is also a background value which represents a value when there is no leaf node or tile. Furthermore, one can also set the status of a voxel or tile to active or inactive, which leverage the performance of the tree traversal since only active elements have to be visited when

traversing. This is especially useful for narrow band level set grids - also known as signed distance field (see).

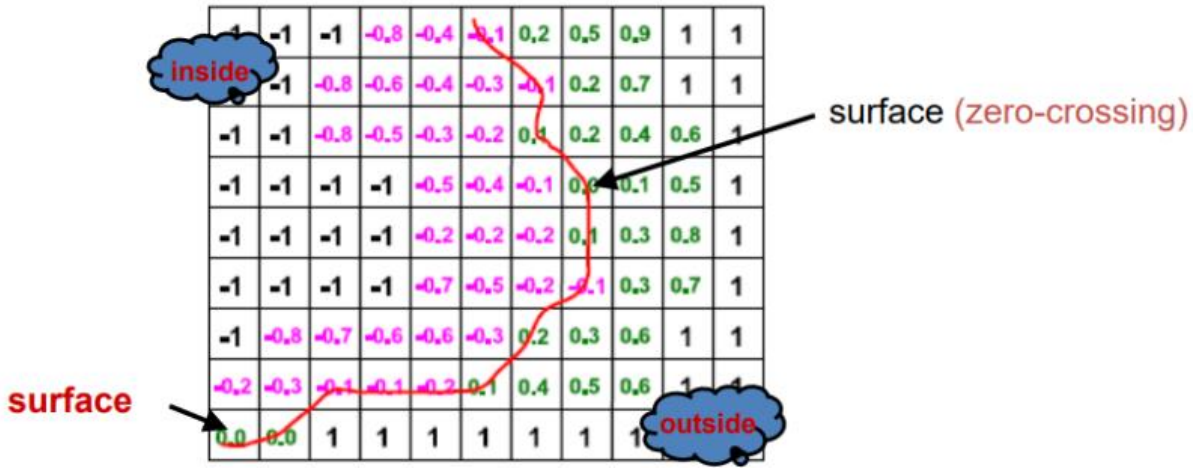

Figure 18: Example of a signed distance field. Each voxel stores the distance from its center to the object surface. Voxels near 0 are surface voxels. Voxels with values greater than 0 are outside the object, while voxels with values smaller than 0 are inside the object. Distance values are truncated at 1 and -1 [60].

OpenVDB is heavily designed for narrow band level sets and fog volumes. There are a lot of classes which provide functions for morphological operation like erosion and dilation, smooth mesh generation, filtering, CSG operations, and more [61]. The morphological functions are especially interesting for the "Virtual dissection" project since a user may want to erode material from the material. There is also stripped-down version of OpenVDB called NanoVDB without all these tools. Although the library is designed for level sets in mind, one can also use it for dense voxel grids. The library also claims to have O(1) complexity when accessing voxels randomly or sequentially. This is achieved by constructing a tree with fixed depth, caching of previous traversals and traversing the three bottom-up whenever possible [57].

5.3. Benchmark

For the performance tests which are required by the "Performance" category of the methodology, a CLI program was created with C++ to automate the benchmarking (see Figure 19).

```
> .\BenchmarkCppVoxelLibraries.exe --help
Benchmarking command line tool.

USAGE: BenchmarkCppVoxelLibraries [OPTIONS]

OPTIONS:

-h -help --help --usage                 Display usage instructions.

-gridXYZ --gridXYZ              Grid size (-gridXYZ 20,20,20).

-cSize --cSize          Cube size (dimension) to delete/add/modify.

-lib --lib              Library to benchmark (cubiquity, openvdb, voxelpl
ay, vdvolume).

EXAMPLES:

BenchmarkCppVoxelLibraries -lib cubiquity -cSize 16 -gridXYZ 20,20,20
```

```
>  .\BenchmarkCppVoxelLibraries.exe -lib vdvolume
Using default cube size of 64 for benchmark...

No grid size specified. Using default size x: 512 y: 176 z: 348
Using "vdvolume" library / libraries.
Executing 'VDVolume' benchmark...
Starting Unity application...
Received cSize: 64
Received grid size x: 512 y: 176 z: 348
Create time (sec): 2.275017
Delete time (sec): 0.1236043
Add/Undo time (sec): 0.1459801
Modify time (sec): 0.01543808
Remove cube parallel time (sec): 0.006285429
Add cube parallel time (sec): 0.006026983
Finished 'VDVolume' benchmark...
```

Figure 19: Benchmarking command line tool for executing automatic benchmarks for all selected libraries. Top: Output to tell users how to use the program. Bottom: Example output.

The `gridXYZ` argument defines the dimensions of the grid, `cSize` defines the dimension of the cube that will access voxels, and the `lib` argument defines the library for testing. For all performance tests, a cube of dimension 64 and a grid of size 512x176x348 (same as in the existing prototype) was used. For all operations, random access was utilised. This is because random access belongs to the minimum set of functionalities the libraries provide. For example, Cubiquity does not have sequential access to voxels, and also does not support multi-threading. Sequential access and multi-threading certainly would make everything faster. It should be also noted that all performance tests were created in C++ to create an equal environment and thus comparable tests (OpenVDB only comes in C++). Since Voxel

Play is a Unity plugin and has no standalone C++ library, the executable of a Unity project build using Voxel Play is called from the benchmarking tool.

5.3.1. Cubiquity

For the benchmark of Cubiquity, the standalone C++ source code was downloaded and built with Visual Studio and CMake. The resulting .DLL and .lib files were included in the CLI program to get access to Cubiquity's functions. Time is measured by the function `std::chrono::steady_clock::now()` for all performance tests. For the first one, the initialisation of a grid, the grid is initialised with `cuNewEmptyColoredCubesVolume(…)` to create a `ColoredCubesVolume` (same as in the existing prototype). The original Unity C# code passes a chunk size (also referenced as base node size in the source code sometimes) of 32 to this method. A user cannot change this value because the value is set to `protected` inside `VolumeData.cs`. One can only try to change this value by altering the source code. Since this is not intended and is also not changed in the prototype, the value is unchanged. In the main loop, the grid is filled with `cuSetVoxel(…)` which creates a voxel at a position in global index space with a custom colour. In Cubiquity you have to pass the colour with Cubiquity's own colour struct `CuColor` which holds the colour converted as a bit-mask. To convert the wanted RGB colour to a `CuColor` the method `cuMakeColor()` is called. The delete, add and modify operations (the next three performance tests) are also using the method `cuSetVoxel(…)`. Cubiquity does not have a delete or modify method. If you want to delete a voxel, you have to set the alpha value to 0, and if a voxel should exist, the value has to be 255. In fact, the code comments of the Unity C# code in `QuantizedColor.cs` say that you should set alpha only to 0 or 255. Since all these three operations use the same method, the performances of these three operations are expected to be basically the same. Except for the delete operation, the reason being that no new random colour has to be calculated with `cuMakeColor(…)`. When deleting a colour, a pre-defined colour with an alpha value of 0 is passed to the method. Interestingly, in the result in Table 5, you can see the cost of the sum of the random colour calculations. While the add operation takes 0.283 seconds, the delete operation takes 0.295 seconds. That concludes that the calculations for new random colours take 12 milliseconds in total.

5.3.2. Voxel Play 2

The benchmark for Voxel Play 2 was created inside Unity since the plugin has hard dependencies to Unity and has no standalone C# or C++ components. To make the benchmarks work with the CLI tool, the application is built to an executable file. The executable file will be called by the CLI tool with the argument `-batchmode` which disables the output of the Unity application and ignores all possible interactable pop-up messages. The grid size and cube size are also passed from the CLI to the Unity application. For time measuring the property `UnityEngine.Time.realtimeSinceStartup` is used. The initialisation process of the grid fills the grid with `VoxelPlayEnvironment.VoxelPlace(Vector3d p, Color c)`

which creates a voxel at position `p` in world space and tints it with colour `c`. In the Unity editor inspector of the volume, the chunk size is set to 16, which is the default value. A quick test showed that with the same grid and cube size used in the CLI, the "create" operation takes approximately one second longer when setting the chunk size to 32. The "delete" operation uses `VoxelPlayEnvironment.VoxelDestroy(Vector3d p)` and the "add" and "modify" operation uses also `VoxelPlayEnvironment.VoxelPlace(…)`. There is no dedicated modify function to change a voxels color, so the same method as for the "add" and "initialise" operation is used. Because of this, the benchmark results, as seen in Table 5: Benchmark results of Cubiquity, Voxel Play and OpenVDB.Table 5, of the "add" and "modify" operation are the same. There are also faster methods to fill a grid. The method `VoxelPlace(…)` has to figure out the chunk the voxel lives in and the local index in that chunk. If you know these informations, you can take the faster overloaded method `VoxelPlace(List<VoxelIndex> indices, VoxelDefinition voxelType, Color tintColor)` where `VoxelIndex` is a struct which contains the chunk and local index information. If you make a ray cast with `VoxelPlayEnvironment.RayCast(…)` you will get this information inside the `VoxelHitInfo` struct. One can also hide a voxel with `VoxelSetHidden(…)` (however, colliders are not affected), check for collision with a collider with `VoxelOverlaps(…)` or add custom data to voxels with `VoxelSetProperty(…)`. One has also to know that world space coordinates in integer match with global index space coordinates because Voxel Play 2 sets the size of a voxel to 1x1x1 in world space coordinates. Voxel Play 2 does not allow it to change the size of the voxels because of performance reasons [62].

5.3.3. OpenVDB

To create the benchmark for OpenVDB, the source code was downloaded and built with CMake. For the grid initialisation, a grid of type `openvdb::Int32Grid` was created and filled via an `openvdb::Int32Grid::Accessor` with the method `setValue(xyz, color)`. An accessor provides random access to the voxels of a grid. When deleting, the voxels were set to an inactive status, and when adding, the voxels were set active again. For the "modify" operation the voxel values were set with the method `setValue(xyz, color)`. For every operation `std::chrono::steady_clock` was used for time measuring. This benchmark mirrors the performance when using OpenVDB with a regular dense grid. For a narrow band level set voxel grid, one would have to change the signed distance field values and activate and deactivate voxels within the modified narrow band along the surface.

5.4. Result

After selecting the libraries and making the performance tests, the scoring for each category has been made. In Table 1 one can see the scores for the category "Functionality". Cubiquity scored with 1.8, Voxel Play with $1.8\overline{6}$ and OpenVDB with $1.5\overline{3}$. In the case of Cubiquity, the functionalities which scored with 1 point would require a lot of custom work to make it possible and are not out of the box supported. Voxel Play scored the highest, however, during the comparison process, the need for resizing the voxel size arose and was added to the minimum compatibly set. Voxel Play does not support resizing voxels and there should not be considered further anymore and received the score 0 for this criterion.

	Cubiquity	Voxel Play	OpenVDB
Functionality	**Score** (min: 0, max: 2)		
Key functionalities			
Initialise a fixed sized voxel grid	2	2	2
Add voxels	2	2	2
Delete voxels	2	2	2
Modify voxels	2	2	2
Needed for tool implementation			
Ray cast against a voxel	2	2	1
Voxel collision with a custom mesh	2	2	1
Needed for rendering			
Mesh generation	2	2	1
Shader to represent simple color	2	2	1
Needed for performance			
GPU instancing	1	2	1
Multi-threading	1	2	2
Optimized grid representation	2	2	2
Occlusion culling	1	2	1
Mesh optimization	2	2	1
Others			
Disk serialisation	2	2	2
Change of the voxel size	2	0 (see 5.3.2 Voxel Play 2)	2
Final score	1.8	$1.8\overline{6}$	$1.5\overline{3}$

Table 1: Comparison result for the "Functionality" category.

The reason for the scoring of OpenVDB is primarily due to the fact that it is a data structure library only and not a Unity plugin. Thus, does not provide Unity integrated rendering

functions. One has to add custom code to make rendering possible. The same is true for the 'Ray cast against a voxel' functionality. With a regular dense grid, ray casting is only supported to leaf node level but not to the actual voxels (see notes on the documentation of the `openvdb::tools::VolumeRayIntersector<…>` and `LevelSetRayIntersector<…>` class) because of performance reasons as Ken Mueth stated in the OpenVDB forum on Google Groups [63].

In the "Usability" category, Cubiquity scored highest with $1.\bar{6}$ (see Table 2). It scored 1 point for the usbality metric 'Flexible' since it provides methods for building a grid from images and data formats from other voxel software. It is also more bare bone functionalities-wise than Voxel Play, which is heavily focused on providing a library to build Minecraft clone. Voxel Play scored in this metric with 0 since you cannot build a grid with any other file format and is, as mentioned, heavily focused to achieve one thing. OpenVDB scored in this metric with 2 since is it primarily just a data structure library without any integrated rendering features for external application like Unity. Cubiquity and Voxel Play are both easy to learn and easy to use. That comes from rich documentation, simple APIs, simple data structures and a lot of 'how-to's. On the other hand, OpenVDB's API and data structure are more complex and the documentation is sparse.

	Cubiquity	Voxel Play	OpenVDB
Usability		**Score** (min: 0, max: 2)	
Easy to use	2	2	1
Easy to learn	2	2	1
Flexible	1	0	2
Final score	$1.\bar{6}$	$1.\bar{3}$	$1.\bar{3}$

Table 2: Comparison result for the "Usability" category.

In the "Support" category (see Table 3) Voxel Play and OpenVDB received a score of 1 for the 'Customer support' criterion since there is no dedicated support team for answering questions for users.

	Cubiquity	Voxel Play	OpenVDB
Support		**Score** (min: 0, max: 2)	
Customer support	0	1	1
Training quality	0	0	0
Final score	0	0.5	0.5

Table 3: Comparison result for the "Support" category.

However, for both libraries, there are forums where one can ask questions and receive answers from the (usually busy) developers or from the community. Cubiquity scored 0 for this metric since it is no longer supported. One can, however, reach out to the original

developer over Reddit. He is quite active on the "VoxelGameDev" subreddit and even responded to one of my questions. For all three libraries, there are no training offerings provided by the developer, organisation, or company.

In the "Documentation" category, Cubiquity and Voxel Play scored the highest with 1.75 (see Table 4). Cubiquity's documentation exists in form of a Doxygen documentation (which has to be built manually) and code comments. Both are rich, have how-to's and cover everything you want to know from a user perspective. The same is true for Voxel Play. There, the documentation exists in form of a website. For both Cubiquity and Voxel Play, you can find everything easily. They provide images which explain functionalities visually, and in case of Voxel Play there are even videos. The documentation for OpenVDB comes in form of a Doxygen documentation hosted on a website. It is sparse and is not even complete at some places displaying "To be written". The lack of rich how-to's can be compensated by looking at the unit tests of OpenVDB in the source code. There are in general no images since the library has no front-end. However, in the original paper, one can see some images of the data structure OpenVDB uses. The documentation refers to the well-written paper in several points. There is also documentation of classes and methods with descriptions. The organisation of the content of the documentation is spread out and uses a lot of cross references. A more centralised documentation would help a user. The language of all libraries is fine. OpenVDB uses a lot of terms which are not immediately clear, but are at some point explained or referenced. In case of Voxel Play and OpenVDB, there is also a FAQ which is useful for a user.

	Cubiquity	Voxel Play	OpenVDB
Documentation		**Score** (min: 0, max: 2)	
Organisation	2	2	1
Language	2	2	2
Graphics and illustrations	2	2	1
Completeness	2	2	1
Final score	2	2	1.25

Table 4: Comparison result for the "Documentation" category.

In the "Performance" category, OpenVDB has overall the best performance closely followed by Voxel Play. The results can be seen in Table 5. The time values are averaged over multiple runs to smooth out the system noise. Cubiquity is by far the worst and is approximately 6 to 7 times worse than Voxel Play and OpenVDB.

Operation/Library	Cubiquity	Voxel Play	OpenVDB
Initialise (seconds)	39.139	6.29	5.85
Delete (seconds)	0.283	0.115	0.041
Add (seconds)	0.295	0.05	0.041
Modify (seconds)	0.295	0.05	0.048

Table 5: Benchmark results of Cubiquity, Voxel Play and OpenVDB.

The worst time is mapped to the score of 0 while the best timing is mapped to the score of 2 as seen in Table 6. Therefore Cubiquity scores 0, Voxel Play 1 and OpenVDB 2.

	Cubiquity	Voxel Play	OpenVDB
Performance		**Score** (min: 0, max: 2)	
Initialize	0	1	2
Delete	0	1	2
Add	0	1	2
Modify	0	1	2
Final score	0	1	2

Table 6: Comparison result for the "Performance" category.

The final result of the library comparison can be seen in Table 7. If you compare the unweighted score, Voxel Play has the highest score closely followed by OpenVDB. But because the emphasis of the library selection is on performance improvements, the performance category got weighted by multiplying it with 2. Thus OpenVDB scores with $3.44\overline{6}$, Voxel Play with $3.07\overline{9}$, and Cubiquity with $2.1\overline{3}$.

	Cubiquity	Voxel Play	OpenVDB
Category		**Score** (min: 0, max: 4)	
Functionality	3.6	$3.7\overline{3}$	$3.0\overline{6}$
Usability	$3.\overline{3}$	$2.\overline{6}$	$2.\overline{6}$
Performance	0	2	4
Support	0	1	1
Documentation	4	4	2.5
Unweighted Score	$2.18\overline{6}$	$2.67\overline{9}$	$2.64\overline{6}$
Final score	$2.1\overline{3}$	$3.07\overline{9}$	$3.44\overline{6}$

Table 7: Final result of the library comparison. For the final score the "Performance" category is weighted by multplying it with 2.

It should be also mentioned that Voxel Play shouldn't be on the list and not considered at all since it cannot provide the function to resize voxels which is included in the minimum

capability set of functionalities. However, the need for this functionality arose during the end of the comparison process and only for the sake of completeness is displayed here.

5.5. Conclusion

In conclusion, OpenVDB should be considered for implementation since it has the highest score. At the end, however, it was decided to create a custom voxel engine for the reasons stated below. Because of that and the discarded "Voxel Play" library, the comparison unfortunately turned out not the be effective. However, much has been learned about the exact requirements for the implementation during the comparison process. For instance, defining a minimum capability set of functionalities helped a lot.

After deciding to go with OpenVDB, a simple C interface was created for the first steps of the implementation with OpenVDB, which can be found in the source code. It created a simple grid with random values and sent the vertices, indices and colour values to Unity using pointers to the data arrays (see Figure 20).

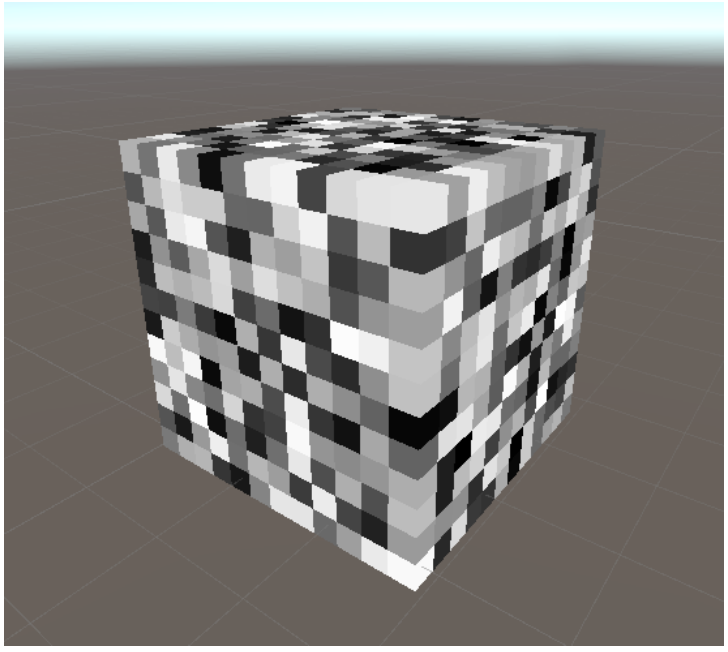

Figure 20: A OpenVDB demo volume rendered in Unity.

This was made by using OpenVDB with a regular dense grid, since using a level set grid creates a lot of unknown problems which would take too much time to research at this point of the thesis. That is because you would have to maintain the distance field values on the narrow band by yourself when modifying the volume (e.g. when cutting or undoing). OpenVDB provides pre-defined functions to modify the volume that are updating these values automatically, but one of the usability problems requires colliding voxels with a custom mesh for more precision. If you want to collide a custom mesh with the volume, you would have to come up with a custom way of updating the distance field values by yourself. Another unknown problem is how this type of working would affect the voxel restoring tool, where you need a collision against removed voxels to re-add them. Therefore, a regular dense grid was used even when one would lose useful OpenVDB functions, like dilation and erosion and accelerated ray casting, with this choice.

After creating a working C interface, the question emerged how one would generate a mesh in an optimal way and how one would ensure fast local mesh generation when cutting for best performance. This is the downside of using OpenVDB with a regular dense grid where you do not have any information about surface voxels for rendering, while with a level set grid you would have by design access to the surface voxels. And also because OpenVDB doesn't have a Unity plugin, you have to come up with such a concept of your own. This fact also reduced the score of the "Functionality" category of OpenVDB when doing the library comparison. The concept that was created is based on defining chunks and flagging voxels with a 'visible' and 'removed' status, which allows displaying only visible voxels. You can all do this with OpenVDB. However, using this concept would mean to use OpenVDB conceptually as a regular 3D array, which could also be done directly in Unity defining a 3D array. Furthermore, the concept benefits from having a direct access to chunks. When using the 'tiles' concept of OpenVDB to fulfil the purpose of chunks, one has no direct access to them. In addition to these facts, you also would get rid of the need for the marshalling process which happens when Unity communicates with the C interface and vice versa, reducing CPU work load and maintenance cost. Because of these reasons, it was decided to go with a pure Unity implementation.

6. Implementation

As mentioned in the last chapter, a custom voxel engine written in Unity is implemented to solve the usability problems and to achieve faster rendering results. The implementation only renders visible voxels and groups voxels in chunks with no tree data structure. The main difficulty when implementing a voxel engine for modifiable CT scans is to achieve high performance while supporting a maximum volume resolution. Thus, the implementation must be able to render and modify millions of voxels while allowing interactions in real time.

6.1. Project setup and structure

The implementation was developed with Unity 2019.4.12f1 and uses following additional Unity packages:

- Entities *(Version 0.11.2-preview.1 – September 17, 2020)*
- Test Framework *(Version 1.1.16 – July 27, 2020)*
- Code Coverage *(Version 1.1.1 – December 20, 2021)*
- TextMeshPro *(Version 2.1.1 – July 27, 2020)*

The source code inside "Assets/VDVolume/Scripts" is structured in the following way:

- **Benchmark**: Scripts for benchmarking.
- **Examples**: Example scripts.
- **Tests**: Edit mode and play mode tests.
- **VDVolume**
    - **Data structure**: Dense and sparse storage of voxel data.
    - **Debugging**: Debugging tools.
    - **Editor**: Unity editor menu and window for exporting the project, creating a `VolumeInitData` object and importing image slices.
    - **Model**: Models for grid and voxel data.
    - **Modifiers**: Code for cutting, restoring, filtering and undoing voxels.
    - **Rendering**: Code for rendering voxels with GPU instancing.
    - **Serialization**: De-/Serialization of a `.vdvolume` file.
    - **Unity components**: Components which can be used in a Unity scene (e.g. `Volume` or `VolumeCollidable`).

Used images slices and generated volume files can be found inside "Assets/StreamingAssets/VDVolume" while other resources are located in "Assets/VDVolume/Resources" and scenes in "Assets/VDVolume/Scenes".

6.2. Concept

The basic concept of the implementation from the perspective of a user can be seen in Figure 21.

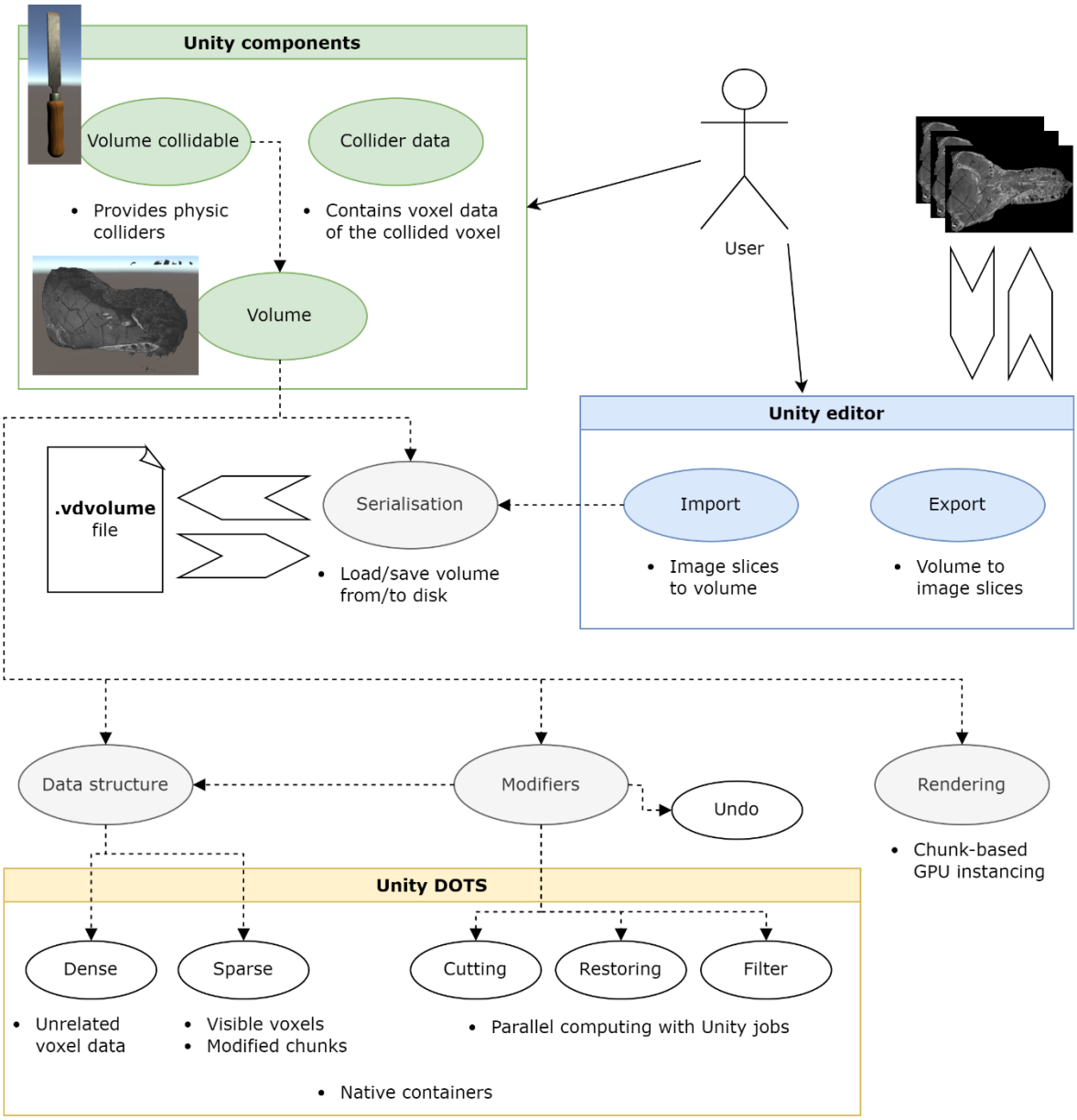

Figure 21: Basic concept of the implementation from the view of a user.

A user has access to `VolumeCollidable` script that can be applied to any object (e.g. a chisel) which should interact with the volume (see section "6.7. Physic collider"). When listening to the Unity's collision events, you then can get the `ColliderData` component from the collider which contains the associated voxel data. The primary script is `Volume`, which can be applied

to any game object and renders a volume. A user executes all functionalities through that script. The user can save and load a volume from a file (see “6.10. De-/Serialisation”), or perform modifying operations like cutting (see “6.5. Modifiers”). There are also events registered on a `Volume`. For more information, refer to the user documentation (see “6.12. User documentation and unit tests”). Furthermore, a user can also import and export image slices (see “6.9. Importing image slices”). A more detailed explanation of the other components will be given in the next sections.

6.3. Model

A voxel stores its index space position, its 8-bit grayscale colour and its state. A voxel can have three different states:

- **Visible**: A visible voxel is a voxel which is visible from any removed voxel. It also satisfies the definition of a solid voxel.
- **Solid**: A solid voxel is a voxel which is not removed and not visible.
- **Removed**: Removed voxels are voxels which are removed, e.g. by cutting or filtering.

How this looks like in action can be seen in Figure 22.

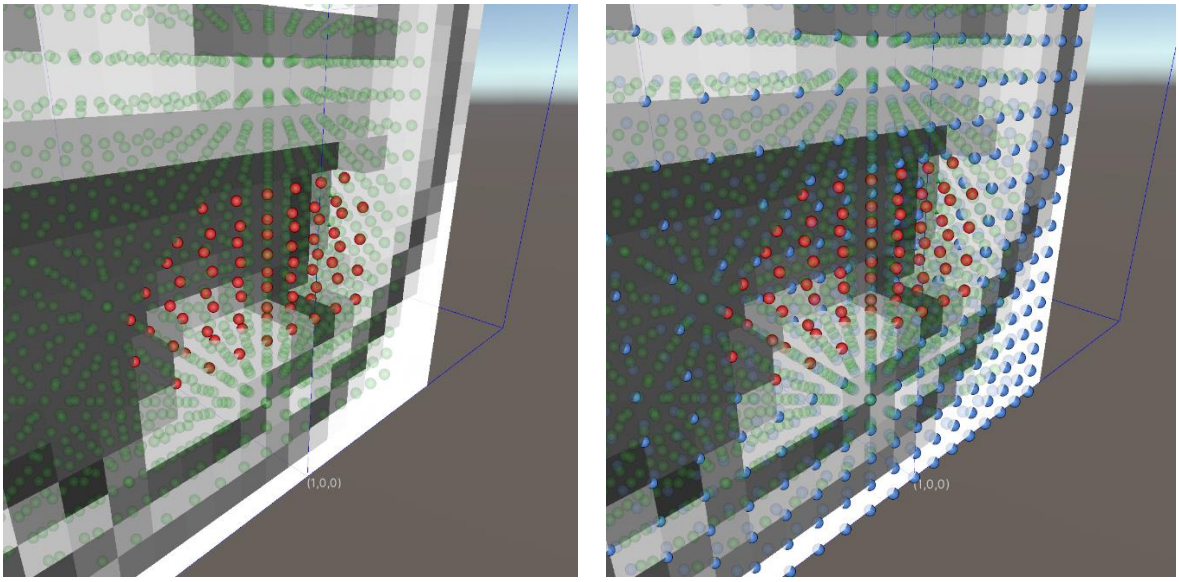

Figure 22: On both images, you can see the same 24x24x7 voxel grid. Left: Only the solid (green dots) and the removed (red dots) voxels are seen. Right: Additionally, the visible (blue dots) voxels are displayed.

6.4. Data structure

Voxels are never fully removed from the data structure. Voxels are densely stored in a 3D array. This array contains all voxels and always stores true data. An octree for the implementation was not considered since Cubiquity also used it with bad results. While an octree can still be applicable when doing ray casting against the volume, or for collision detection, the traversal of the tree is usually slow, especially for dense grid used in this application. The grid has to be dense all the time since the application supports a restoring tool which needs collision detection on voxel level even if the voxels are removed. This leaves the volume to be a dense grid all the time. An octree in this scenario would have the maximum depth and node count all the time. If searching for a voxel, you would have to traverse the tree in its full depth all the time, which makes the operation slow, while an array approach that is divided into chunks delivers direct access to the voxels. Even if you have a non-dense grid implemented with an octree, and divided the volume into chunks with the inner nodes, the operations of adding and removing nodes can become slow.

In this implementation, the chunks are represented by a sparse storage that only stores visible voxels inside visible chunks (see Figure 23). Visible chunks are chunks which contain visible voxels. This sparse storage is made of multiple supporting hash maps. More information about the supporting data structures and for what they are needed can be read in the source code annotations.

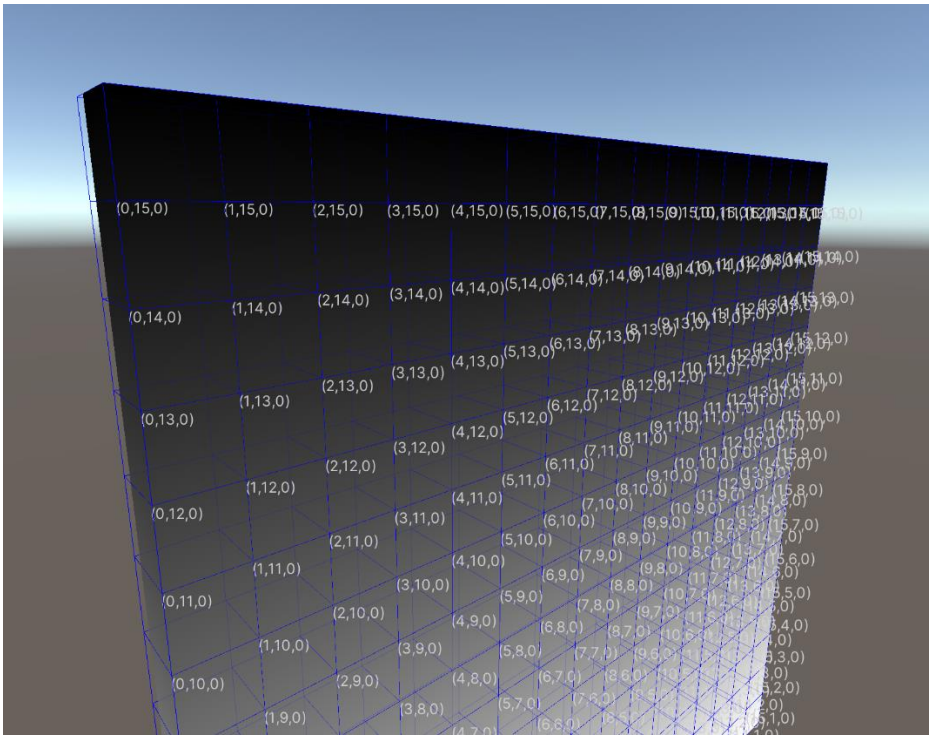

Figure 23: A 256x256x12 voxel grid divided into 16x16 visible chunks. Each chunk can hold 16x16x16 voxels. The white labels indicate their index space position (x,y,z) inside the volume.

A chunk always has the shape of a cube with equal edge length and encloses voxels inside it. Making the chunks smaller or larger only has an impact on updating the compute buffer described in "6.8 Rendering". These buffers are updated when modifying the voxel grid and are mapped to the chunks of the volume. When making a chunk smaller, the application has to iterate over smaller compute buffers, which leads to a lot of I/O operations to the GPU. I/O operations are expensive and should be avoided. When making the chunk size larger, it reduces the number of compute buffers. However, when a chunk is modified by any modifying operation, more voxels have to be sent to the GPU, increasing the time of the I/O operation.

All data structures used in the dense and sparse storage are from the "Unity.Collections" DLL which comes with native containers when using Unity's "Entities" package. This is because the application heavily utilises Unity jobs of Unity DOTS. The dense storage uses a 1D `NativeArray` while the sparse storage uses a `NativeMulitHashMap` and several regular `NativeHashMap`'s. In general, the whole application only uses 1D arrays. 3D positions from 1D indices and vice versa will be calculated at runtime. Starting with (x:0, y:0, z:0), columns will be filled first (in Unity this matches the positive x-direction), followed by rows (in Unity this

matches the positive y-direction) which then is followed by depth (in Unity this matches the positive z-direction). While using native collections is mandatory for using Unity jobs, it also provides fast memory allocation, which is allocated outside of the managed C# runtime. This means the garbage collector won't move the memory around. Through the allocation of a single memory block, the data is also cache coherent, which may result in less cache misses during cutting or restoring. From a future perspective, 1D arrays are also needed when you want to parallelize the application on a GPU since frameworks like Nvidia Cuda or OpenCL only work with 1D arrays. Native containers only work with non-reference types to enforce safety. This is why the data of a voxel is stored inside of a struct. In general, the application avoids creating new C# objects, uses object pooling (e.g. for the box colliders of the `VolumeCollidable` script) and pre-allocates memory whenever possible.

6.5. Modifiers

There are four ways how one can modify a volume: by cutting, restoring, filtering or undoing.

6.5.1. Cutting and restoring

Cutting means to remove a voxel while restoring means to add cut voxels back. When cutting or restoring one has to manage the state of the voxels (dense storage) and the visible voxels list (sparse storage). To illustrate the process of cutting and restoring of a single voxel in a 2D scenario, Figure 24 can be viewed.

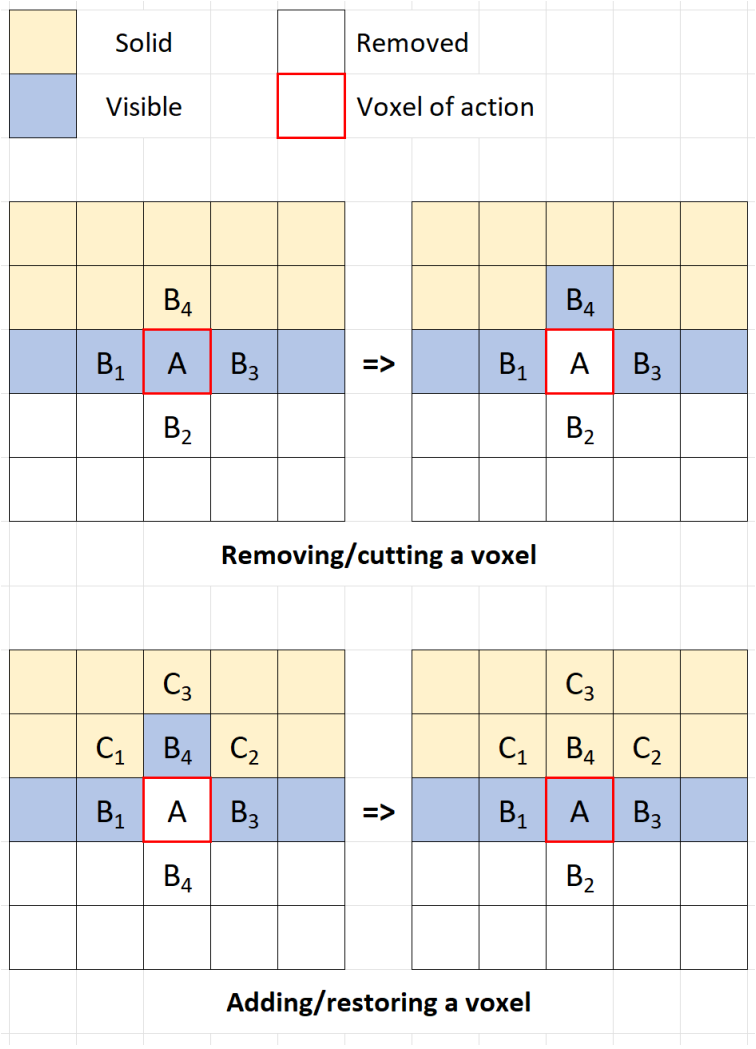

Figure 24: Illustration of the process when removing or adding a voxel.

A voxel A will be only removed if it is not already removed. If that is not the case, the neighbour voxels (B1 to B4) will be checked if they have the state "Solid". If that is the case, that voxel will be made visible. For adding a voxel, the rule set is larger. This is to avoid visible voxel artefacts when re-adding voxels. If you have a look at the left grid at the bottom of Figure 24, you want to make voxel B4 solid when re-adding voxel A. The data would not be wrong if you don't avoid this, but it would create unnecessary voxels to render. Therefore, when re-adding voxel A, the neighbours C1 to C3 of a neighbour voxel B will be additionally

checked. In the case of B4: If B4 is visible and if C1 to C3 are all visible or solid, voxel B4 will be made solid. If one of the neighbour voxels of voxel A (B1 to B4) is removed, voxel A will be made visible. If voxel A is visible (in the case an already visible voxel should be made visible), voxel A will be made solid when all neighbours are solid or visible.

6.5.2. Parallel filtering

Filtering was a requested feature of the application. With filtering, you can remove a value or a value range from the volume. For example, you can remove the bedrock of a scanned fossil (see Figure 25).

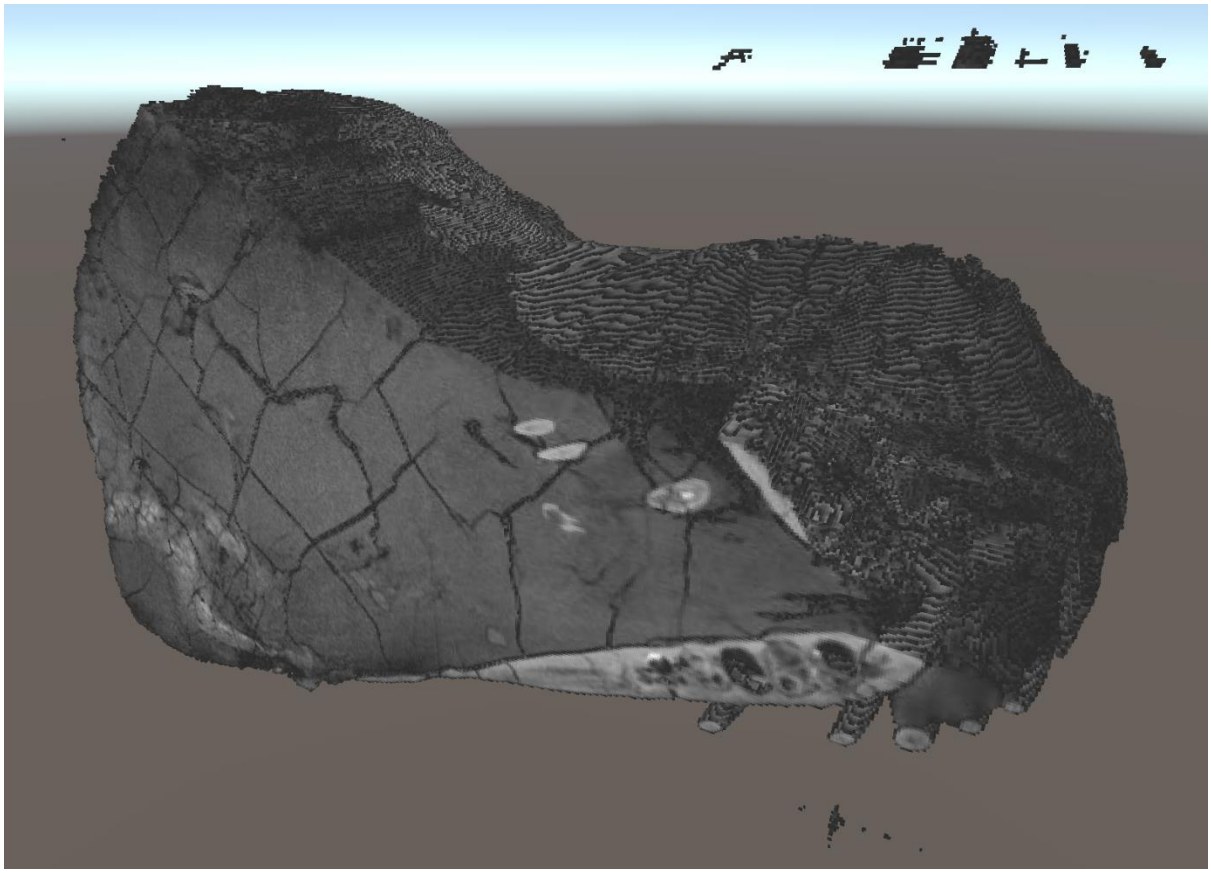

Figure 25: This is the 512x348x176 voxel volume used in the prototype and the benchmarks. Here, the colour values in the range of 0 to 42 are filtered. With this, the bedrock has been removed and only the relevant voxels are left.

To make the filtering as fast as possible, Unity jobs were used. There is also the alternative to use .NET threads instead. However, Unity jobs provide fast thread creation, worker threads will be distributed equally on the available CPU to minimize context switching, threads will be reused to avoid recreation of threads, and the Unity Burst compiler optimizes the code of a job for performance. The filtering process is divided into three phases (see Figure 26).

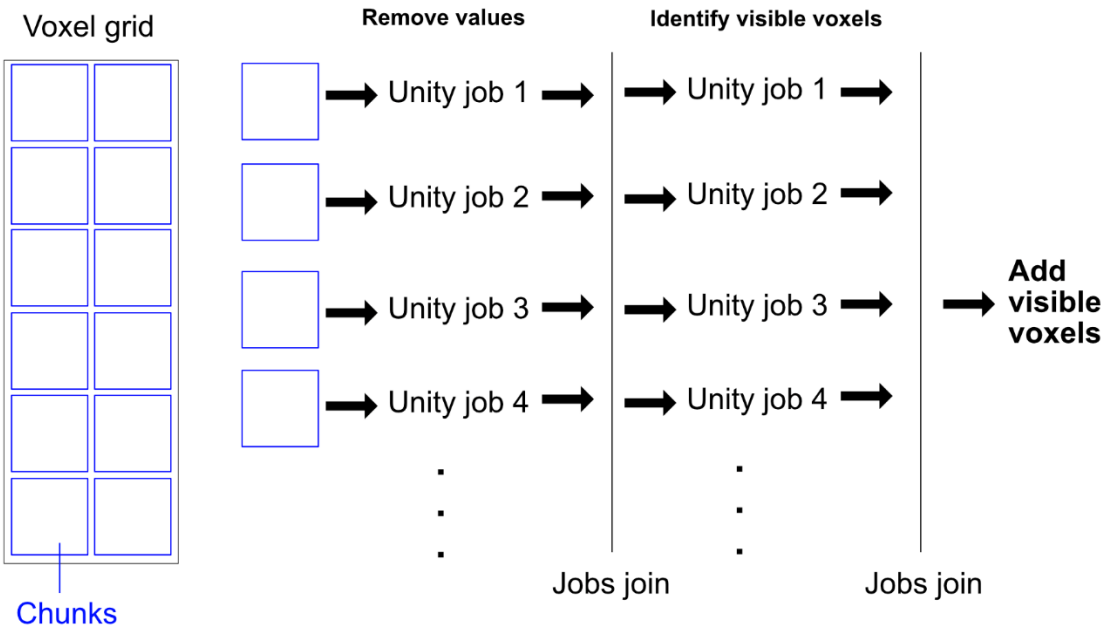

Figure 26: Process when filtering a value or a value range.

Each job works on a chunk. In the first phase, the value/s are removed from a chunk. This is accomplished by modifying the dense storage of the volume. In the second phase, the voxels which should be made visible after the removal are identified. These voxels will be added to the volume in the last phase. Between each phase, the threads will join. Only the first two phases are executed in parallel. Before explaining the reason for the splitting of the phases, an example of how the three phases are affecting the data structure will be explained (see Figure 27).

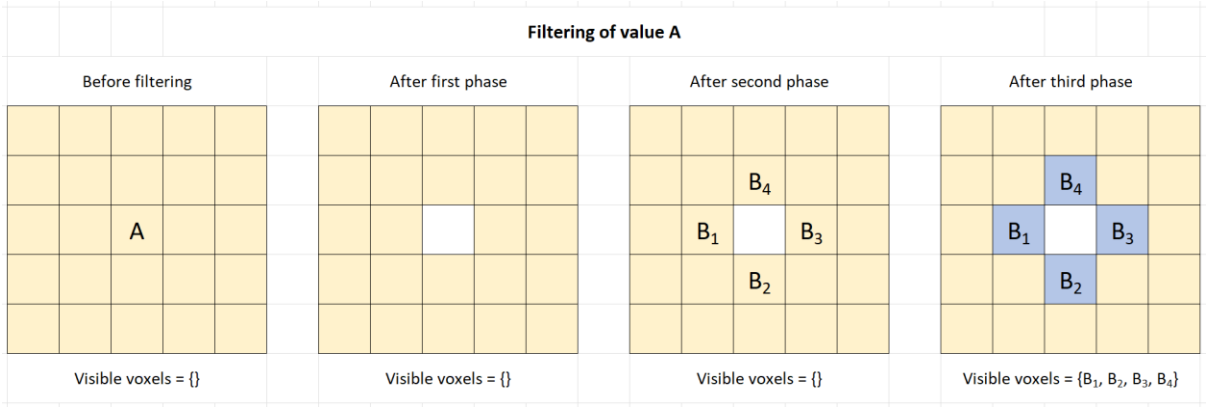

Figure 27: Data structure at each phase when filtering value A. Colouring same as in Figure 24.

In Figure 27, you can see the same portion of a voxel grid four times. On the left, there is a value A which should be filtered. After the first phase, this voxel is removed from the dense storage. Now there is a hole inside the grid which can not be seen since there are no visible voxels around it. To fix that, phase two identifies the voxels which should be made visible (B1-B4). The state of these voxels is then set to “Visible” in the third phase. In this last

stage, the voxels are also added to the sparse storage, which holds the visible voxels which are rendered at the end.

Now for the reason for the phase splitting: The reason for splitting the process in phases is that on the sparse storage data structures you can not work in parallel. The sparse storage uses hash maps which can't be accessed in parallel, since this would create race conditions and result in undefined behaviour. There is a function which provides thread safe access, however, this is accomplished by semaphores which are blocking calling threads. This will lead to a decrease in performance. This is why the phase three exists. The reason for phase two is that you can not identify the visible voxels in the first phase because of chunk edge problems (see Figure 28).

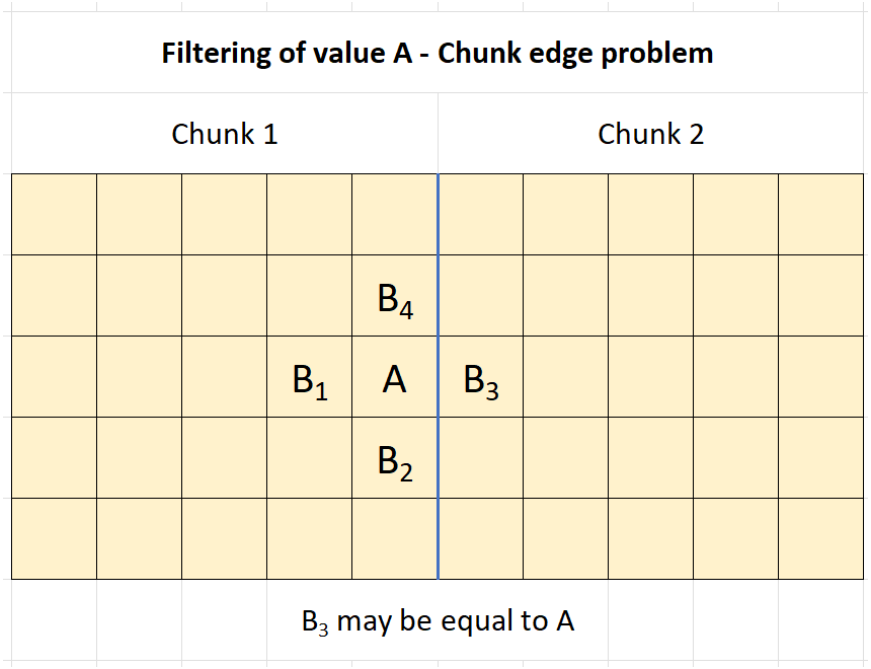

Figure 28: Chunk edge problem when filtering a value A.

Figure 28 shows the problem which can occur when trying finding the visible voxels in the first phase. If a job J1 works on chunk 1 while another job J2 is working simultaneously on chunk 2, J1 can not determine if B3 is a visible voxel. This is because B3 might be removed from job J2. Phase two therefore depends on phase one.

Phase two uses a custom data structure for saving the identified visible voxels (see Figure 29). This is because all jobs have to write in parallel on the same data structure.

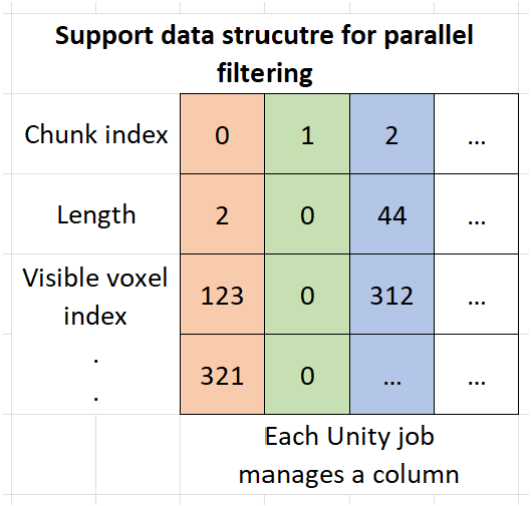

Figure 29: Custom data structure for storing identified visible voxels.

The data structure for storing the identified visible voxels is conceptionally a 2D array (in the actual implementation it is a 1D array). Each column stores 1D indices of identified visible voxels. Each column is mapped to a chunk. This way, all jobs can write simultaneously to the array. To make adding and iterating fast, the lengths of the stored identified visible voxels are stored. Each column is pre-allocated with the amount of voxels inside a chunk. From this data structure, the last phase will get the voxels and add them to the visible voxels list.

6.5.3. Parallel cutting and restoring

For coarse but fast cutting and restoring, a parallel alternative is implemented. This method currently only works with cubes, but a rectangular cuboid or other primitives can be added later. It divides a cube into layers to feed the jobs with separated data (see Figure 30).

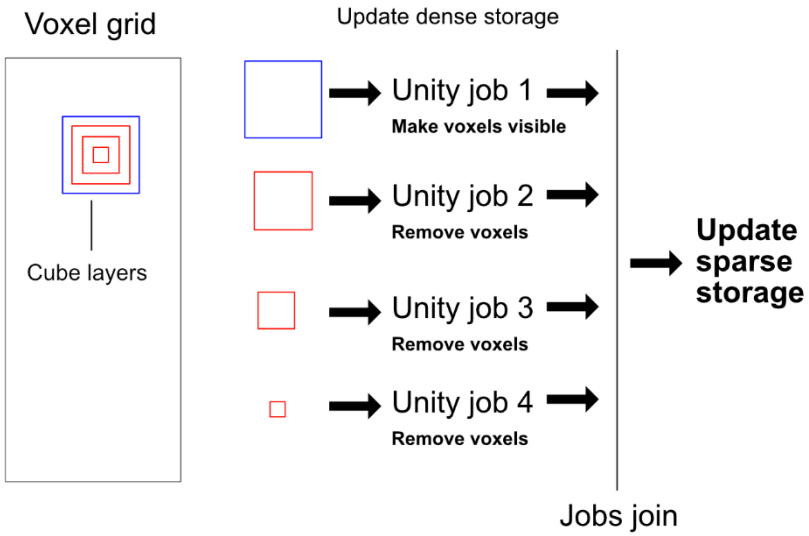

Figure 30: Illustration of the parallel cutting process. Blue: Hull layer. Red: Inner layers.

The voxels in the inner layers are removed, while the hull layer job adds visible voxels to the volume. The process works similarly to the filtering process. When restoring, the inner layer jobs add voxels by making them solid, while the hull layer job makes the voxels visible.

6.5.4. Undoing

The undo functionality allows the user to undo voxels which have been cut or restored in a not parallel way. Removed or restored voxels associated with their action (removed or restored) will be added as commands in a ring buffer having a size of 262.144 (2^{18}). This limit can be adjusted in the code. When a user calls the undo function, all commands will be reversed, beginning with the last added command to the ring buffer and continuing with the most recently added voxels. After this, the buffer will be emptied.

6.6. Ray casting

Ray casting is implemented in a very simple way. How it works can be seen in Figure 31.

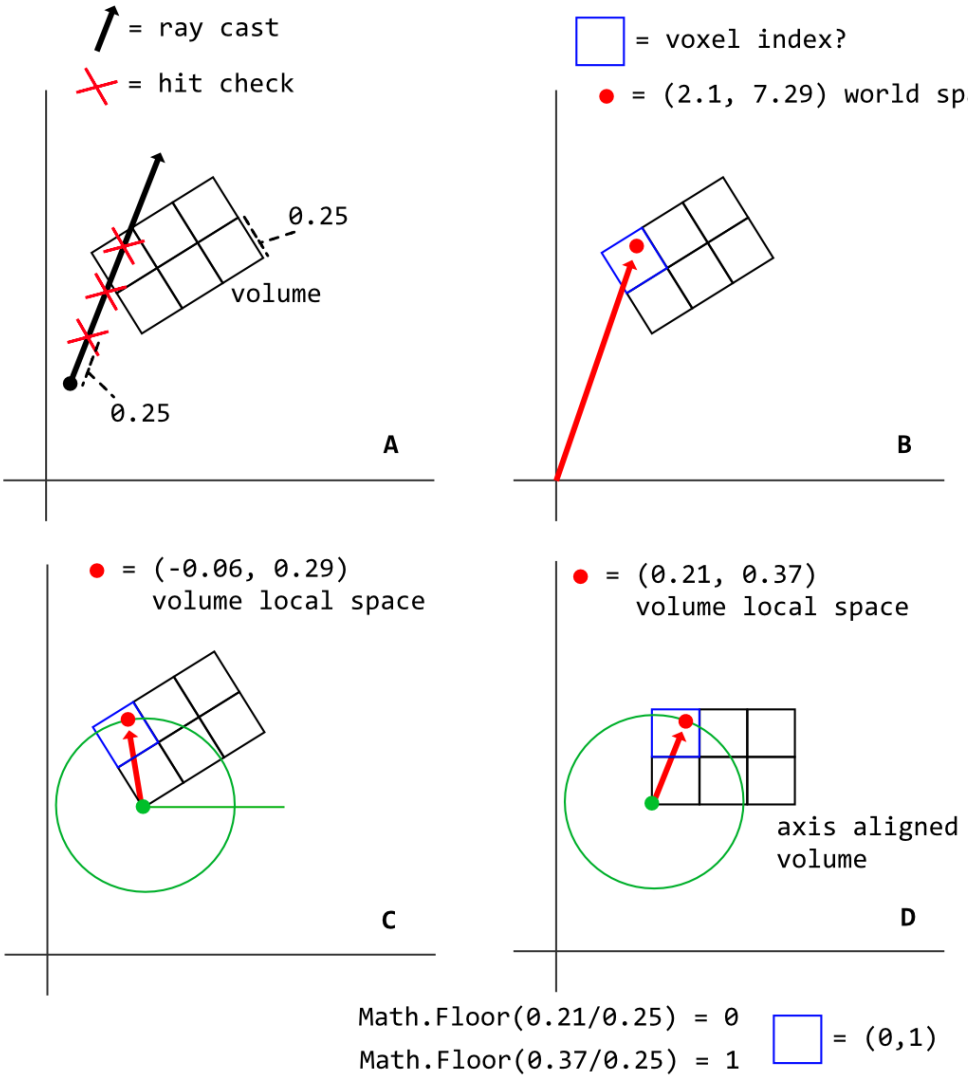

Figure 31: The process of ray casting.

As you can see in segment A, the script will check in a fixed size step if a point is inside a voxel. The length of the ray cast can be adjusted by a parameter. To check if a point in world space (B) is inside a voxel, the script will try to retrieve the voxel by a voxel index coordinate. To get the voxel index, the world space point will be transformed to the local space of the volume (C). The volume has its origin in the lower left corner, which is aligned with the origin of the local coordinate system. After this step, the inverse rotation of the volume will be applied to the point (D). This mimics the situation when the volume would be in an axis aligned pose. After that, the components of the index coordinate will be calculated by dividing each component of the rotated point with the scale of a voxel, followed by applying the `Math.Floor()` function to that result. This implementation has, however, an accuracy problem, which may not be noticeable when the volume resolution and the distance of the camera to the volume is large enough, but still is a problem. For instance, if the ray is

near a corner it will jump over it missing a contact point. How this can be easily improved is described in “8. Future improvements”.

6.7. Physic collider

When cutting a volume, there are two ways one can interact with the volume: by ray casting or with physic colliders for voxels. The prototype, as already mentioned, uses ray casts. This is, like already explained in "4.3.1 Possible improvements", an inaccurate method for cutting with a sphere or any shape because you would need numerous ray casts in all directions to sample the surface of the object. And somehow you would have to come up with an algorithm to delete the voxels which enclose this object. In the case of the prototype, that object was a sphere where it is fairly easy to determine these voxels. For a more accurate and robust solution, the application makes it possible to collide any mesh collider with the voxels of the volume (see Figure 32). A more accurate cutting tool was requested by the prior heuristic evaluation and can only be achieved by supporting physics collider. The only action a user of the application has to take is to put the `VolumeCollidable` script on the game object which should cut the volume. The other scripts on that object can then listen to the `OnCollisionXXX` events from Unity. Voxel data of the collided voxel can be received by getting the `ColliderData` component of the game object of the voxel collider. Further instructions of how to use the `VolumeCollidable` script can be read in the Doxygen user documentation.

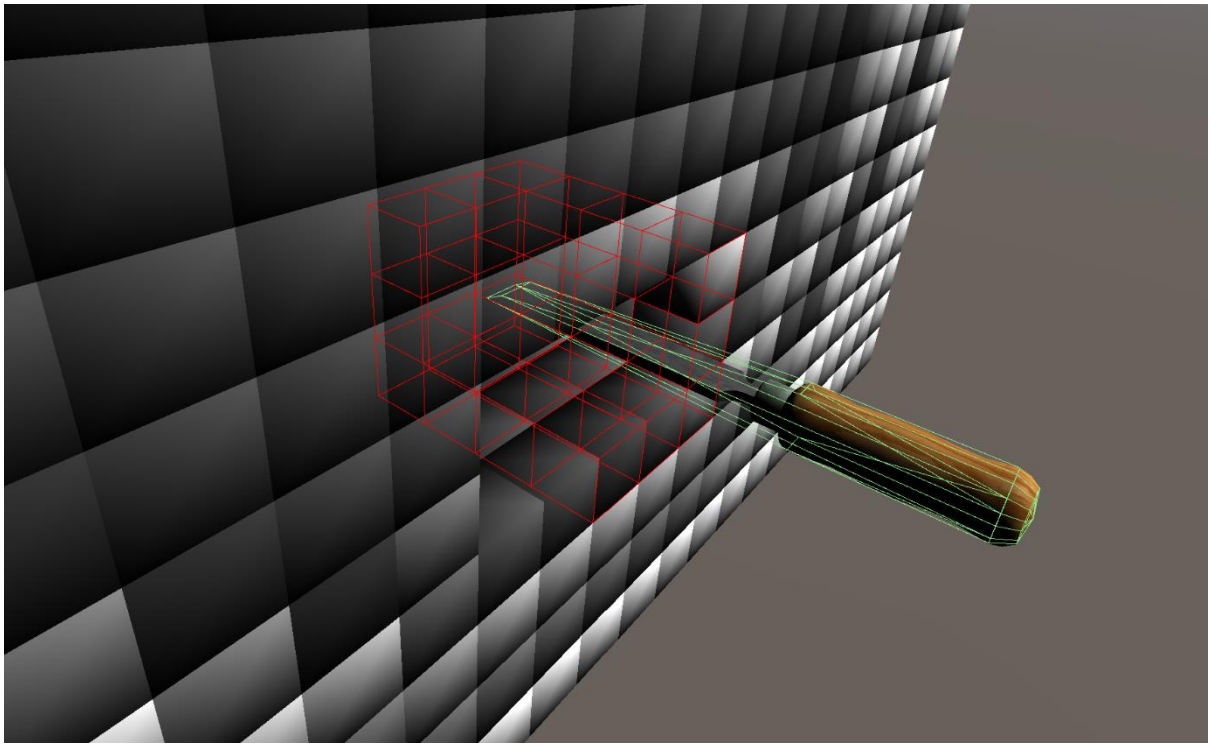

Figure 32: An example of a custom mesh collider (a chisel) cutting a volume.

For this to work, a local-based approach was chosen. Adding colliders for possible millions of voxels is not an option since in the normal way you would have to create game objects for each collider. Having millions of game objects reduces the performance of Unity heavily and Unity is not suited to be used with this huge amount of game objects. For having millions of game objects, Unity introduced Unity DOTS, which works with entities instead of classic game objects. The approach of calculating physics with Unity DOTS is actually one of the

improvements for the application discussed in "8. Future improvements". The current implementation uses colliders with game objects.

To avoid creating an unnecessary number of colliders, the `VolumeCollidable` script checks each frame if there are voxels of the volume inside the AABB (axis aligned bounding box) of the Unity collider attached to the object. The checking is done by stepping with the size of a voxel over the space of the AABB. If there is a voxel, the game object of a `BoxCollider` component will be aligned with the voxel mesh. The game objects of all possible colliders are pre-allocated to create an object pool. If the AABB can enclose a maximum 16 voxels, there will be 16 game objects allocated, including the attached box colliders. When aligning a collider, the script will use a game object from this object pool for better performance. Only box colliders which are aligned with voxels of the volume are enabled, all other are deactivated (see Figure 33).

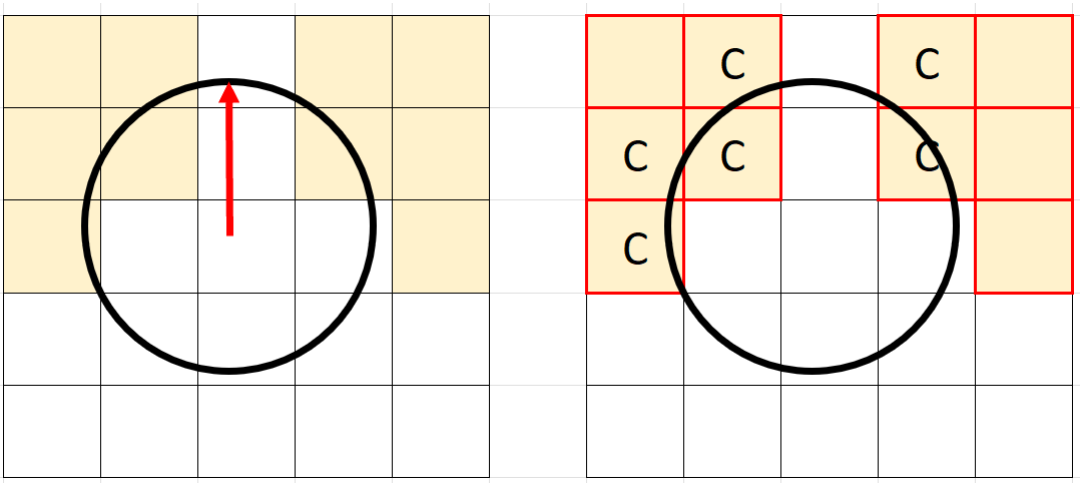

Figure 33: Comparison of voxel collision models. Left: The ray cast method. Right: The collidable method. While the left method does not recognize any detection, the right method will remove all voxels which collide with the sphere mesh. Red voxel borders show enabled colliders and the letter "C" shows collisions. Yellow squares: Solid voxels. White squares: Removed voxels.

Game objects are never deactivated since that would decrease the performance heavily when doing it on thousands of game objects. When aligning a collider, a `ColliderData` component will be added to the game object of the collider. This `ColliderData` holds the voxel data. How the described process looks in a Unity scene can be viewed in Figure 34.

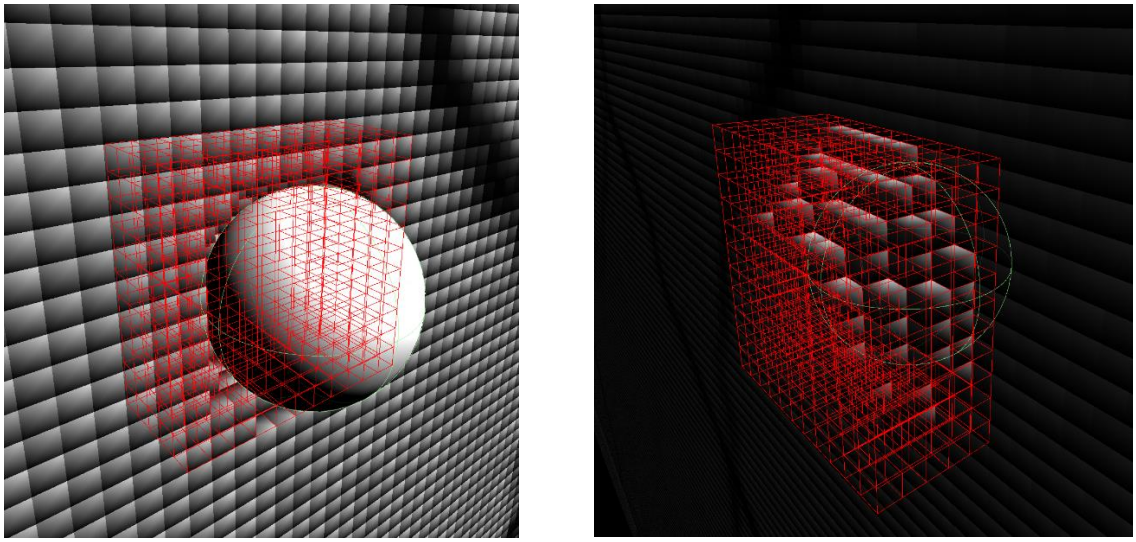

Figure 34: Visualisation of the enabled colliders when cutting the volume with a "VolumeCollidable"-object. Left: The volume viewed from the outside. Right: The same volume viewed from the inside.

Examples for the ray cast and the collider method for cutting the volume can be found in the examples folder of the source code.

6.8. Rendering

The application only renders visible voxels which are stored in the sparse storage. To further improve the rendering performance, GPU instancing is used to render the voxels (see Figure 35).

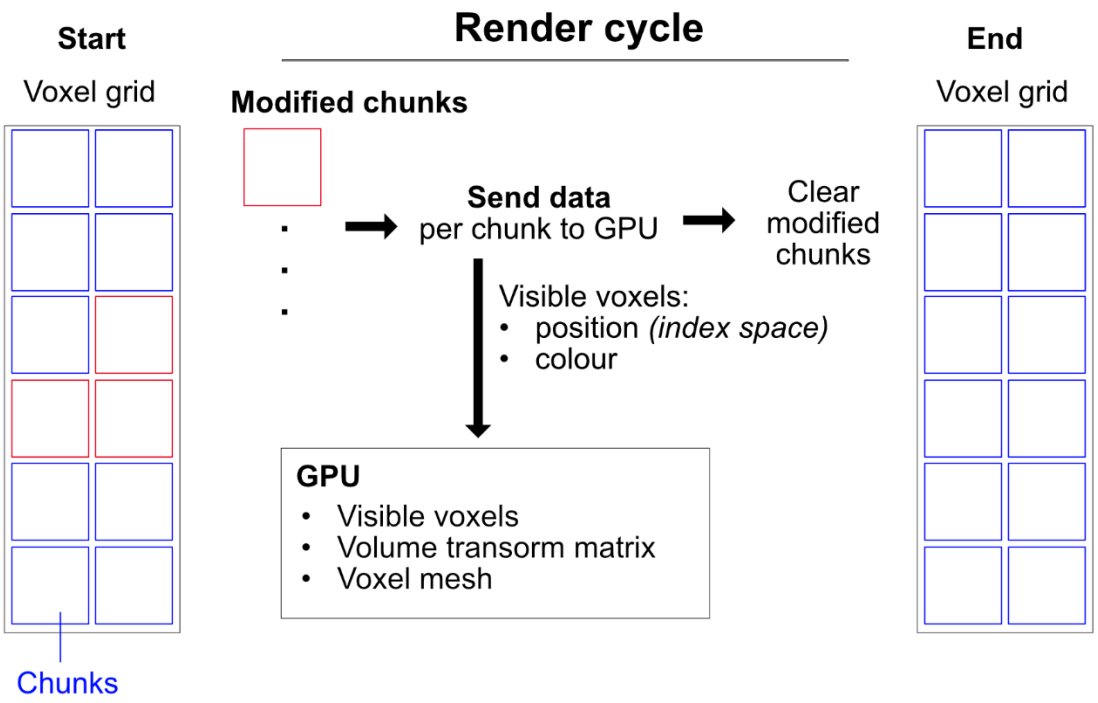

Figure 35: The render cycle of the application.

The benefit of GPU instancing is that less data is sent to the GPU, which is also reducing state switches inside the graphic API. All relevant data for rendering is stored on the GPU. That is: the position of the voxels in index space, the colour of the voxels, the voxel mesh (which has its centre not at the local coordinate system origin but at (x:0.5, y:0.5, z:0.5), and edge lengths of 1) and the transform matrix for transforming the vertices to the correct world space coordinate. The GPU will then draw the voxels with the help of a shader.

When modifying the volume, you don't want to re-upload all data to the GPU since this would not work in real time when working with millions of voxels. This was tested with a side project where five million cubes were repeatedly uploaded to the GPU. Therefore, the application only uploads the data of chunks which were modified in the last frame. This saves a lot of time and adds a concept of locality to the cutting-rendering process. After the uploading is done, all modified chunks are set to be not modified. When initialising a volume, all chunks of visible voxels will be marked as modified.

When using GPU instancing, you disconnect the mesh information from any Unity game object, leaving you managing translation, rotation and scaling by yourself. Because of that, the shader not only renders the voxels, but also translates them to the correct position. This

could have been done on the CPU, but on the GPU it is faster since it can be executed in parallel (not a major performance factor, however). This was achieved by sending the volume transform matrix to the GPU whenever the Unity transform object of the volume changes. This matrix not only transforms the origin of the voxel mesh to the correct world position, but also puts the mesh vertices in relation to that position in the correct coordinate. This is simply done by overwriting the last column in that matrix with the translation vector of the transformed world position of the voxel origin [64]. Without that, only the world position of the voxel origin would have the same rotation as the volume, but not the voxel mesh when rotating the volume.

The application supports full support for scaling, rotation and translation (see Figure 36). This was actually one of the hardest parts to achieve apart from supporting voxel colliders (where the most difficult part was also related to scaling, rotation and translation).

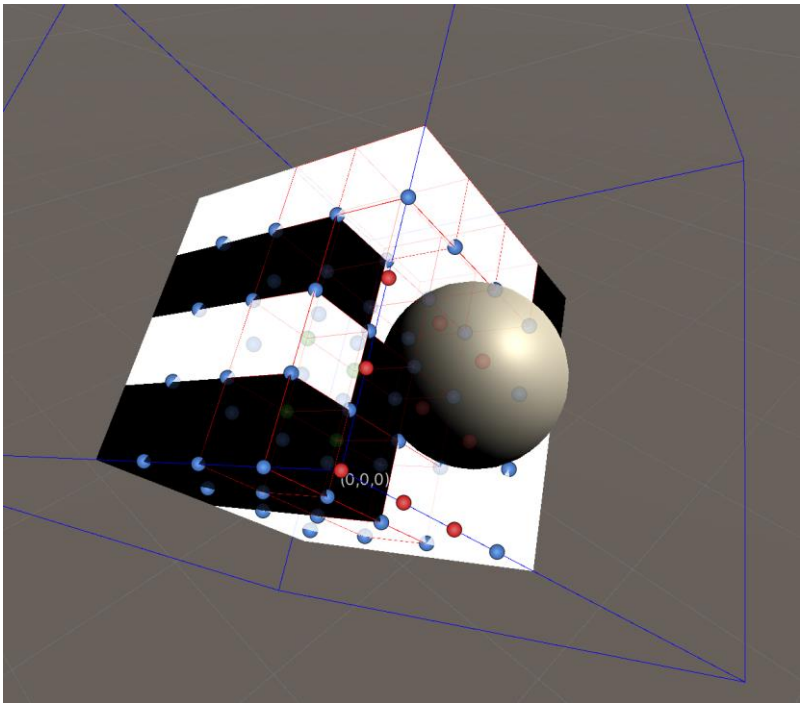

Figure 36: A rotated, translated and scaled volume with debugging visualisation turned on. The volume is cut by a sphere collider.

At the beginning, only the voxel colours without lighting were rendered, which made the volume confusing to look at when cutting voxels (see Figure 37).

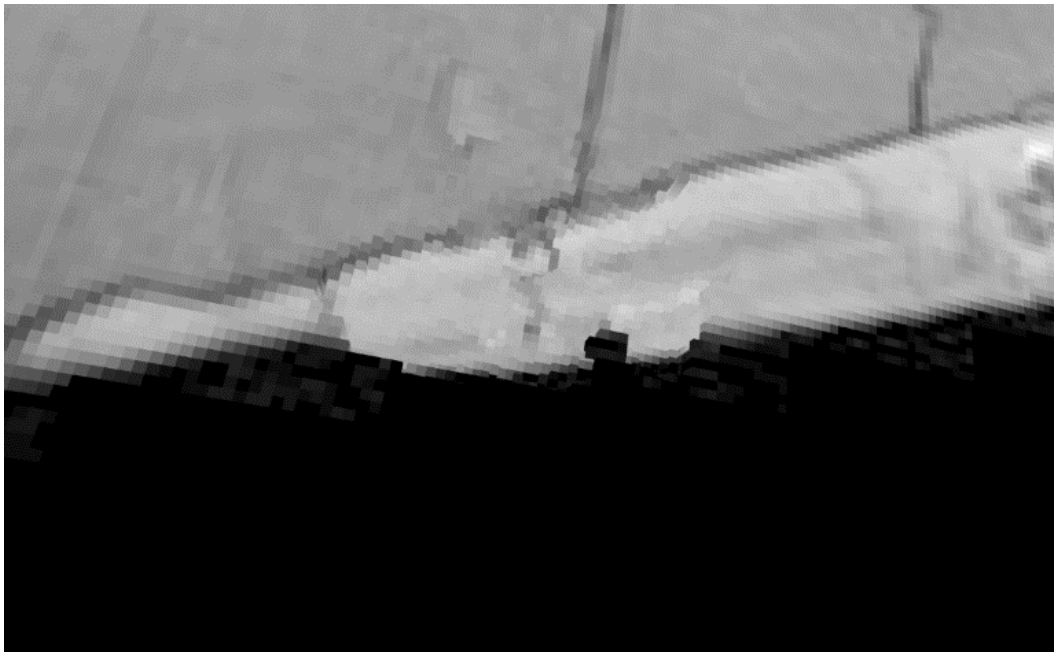

Figure 37: Voxels rendered with only their colour value.

That is because it was hard to distinguish neighbour voxels. Because of that, the shader was changed, now applying simple Phong shading to each individual voxel (see Figure 38).

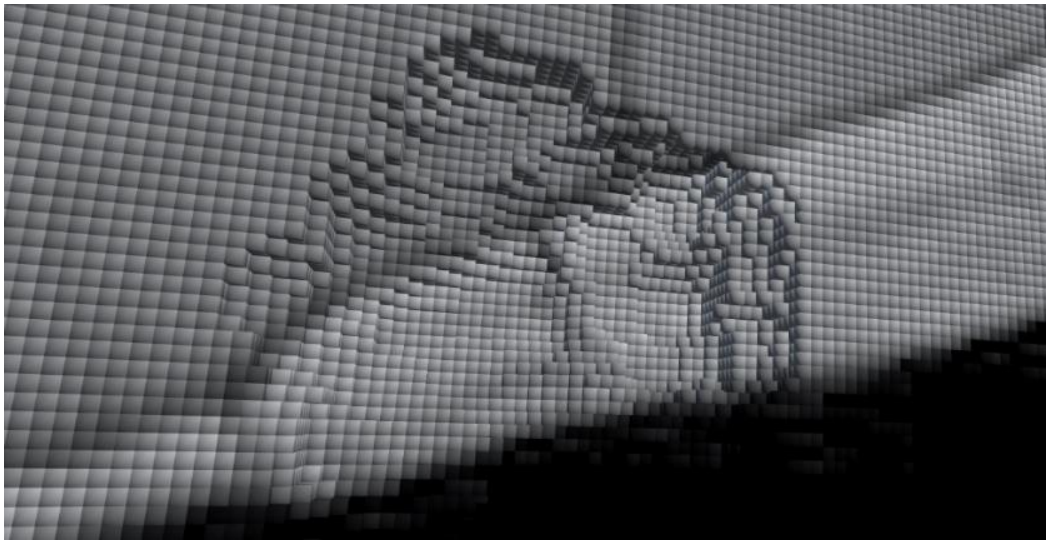

Figure 38: Voxels rendered with Phong shading.

This made it easier to differentiate voxels. However, one could argue that this distorts the original colour values, which may be of importance. The light intensity and colour of the light source in the Unity scene will have an impact on this. This also introduced a screen door effect, which in turn let to a Moiré effect when viewing the image from a distance (see Figure 39).

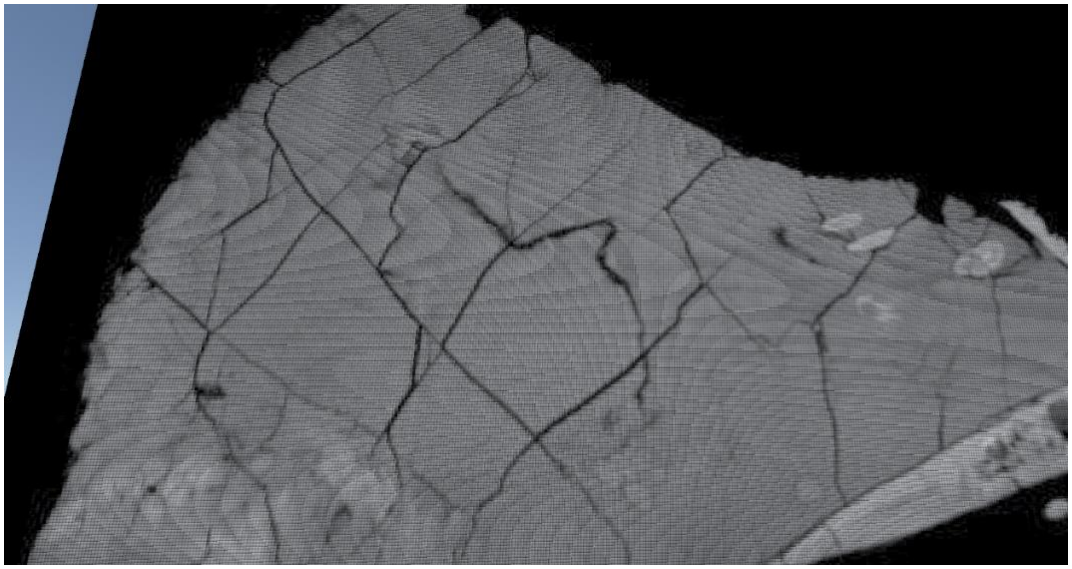

Figure 39: Phong shaded voxels introduced a Moiré effect when viewing the volume from a distance.

To reduce this effect, 8x MSAA was activated, which resulted in a cleaner image (see Figure 40).

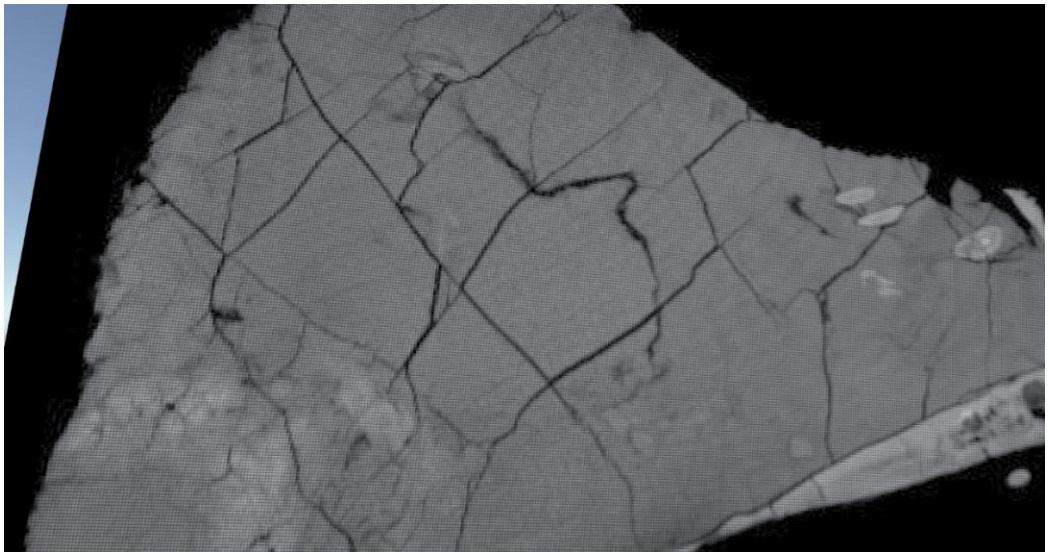

Figure 40: Volume rendered with 8x MSAA activated, reduced the Moiré effect.

Other anti-aliasing methods can be used, which may be better or faster. To do this, one has to download the "Post Processing" package from Unity. After adding a "Post-Process Layer" to the camera, one can choose different solutions.

6.9. Importing image slices

Before you can start working with the application, you have to create volume data by importing image slices. You can access this functionality by a window inside the Unity editor (see Figure 41).

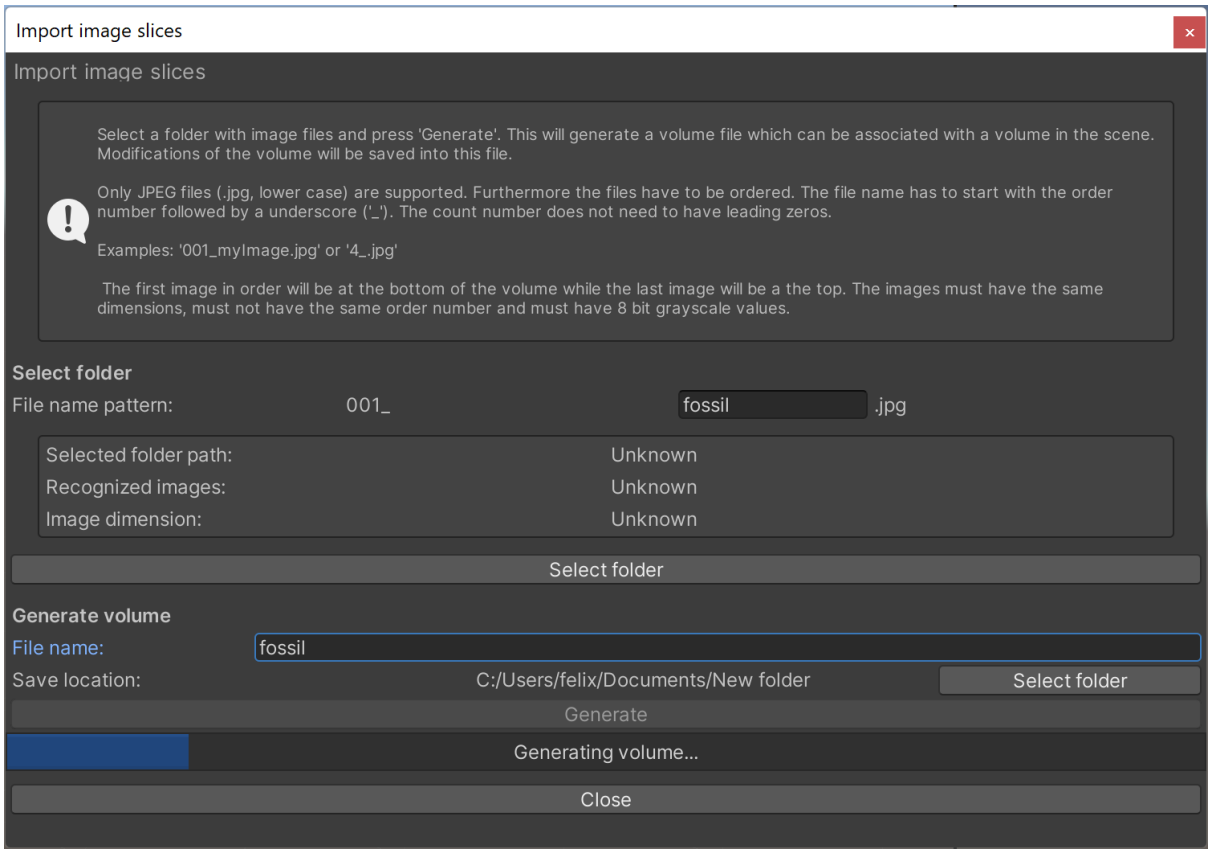

Figure 41: Unity editor window for importing image slices.

Inside the window, which can be opened by the "VDVolume" menu (see Figure 43) you can select a folder with image slices and press "Generate" to create a `.vdvolume` file which contains the voxel data and the dimension of the voxel grid (see Figure 42). The selection of the image folder will return some information about the files. Both, selecting the folder and generating the `.vdvolume` file are asynchronous operations which are executed with a .NET thread. The progress of these operations will be reflected by a progress bar inside the window. The read process will begin with the first image and iterates over the rows of the pixel matrix in reverse order of the image. The row iteration is followed by an iteration over the columns. The first image will be displayed in Unity 's xy-plane standing upright. Following images will be added in Unity's positive z-direction (which goes into the depth). The images you provide to the application need to have the JPEG file format, need to be 8-bit grayscale images and need to be numbered at the beginning of the filename. If you already have a sorted stack of images but with the wrong file name format, you can easily rename them with the command:

```
ls | cat -n | while read n f; do mv "$f" `printf "%03d_fossil.jpg" $n`; done
```

More information about the import process can be read in the Doxygen user documentation and inside the window itself.

The `.vdvolume` file, which is generated by the import process, stores voxel data efficiently by storing the bits which make up voxel data. For each voxel, the 8-bit grayscale color and the 2-bit state is stored in a bit mask of size 10. The position of the voxels is derived from the order they are stored inside the file. The order of the voxels inside the file also matches the order the voxels are stored inside the 1D array of the dense storage. The grid dimensions are integers and take each 4 bytes, which theoretically allows for 2.147.483.647 voxels per axis. In future they can be made unsigned integers if the rendering allows it performance-wise and your computer memory is large enough (see "7.1.4. Limitations").

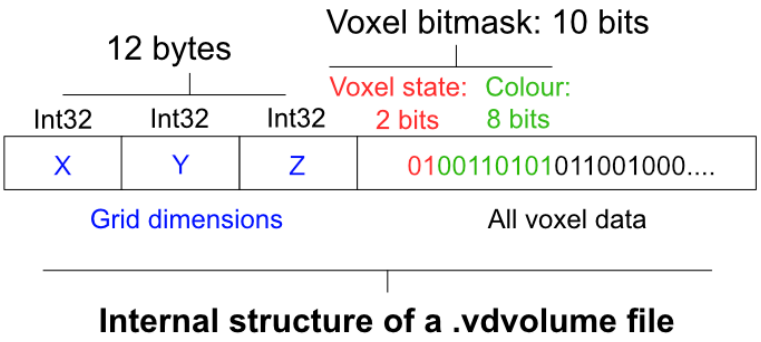

Figure 42: Internal structure of a ".vdvolume" file.

6.10. De-/Serialisation

When working with a volume, a user wants to load and save a volume. When initialising a volume, the application loads automatically a volume. You can pass the volume data to the application by entering a path to the `.vdvolume` file in the volme game object inspector or by creating a `VolumeInitData` object via the "VDVolume" menu (see Figure 43). The `VolumeInitData` is a container for the inittialisation arguments.

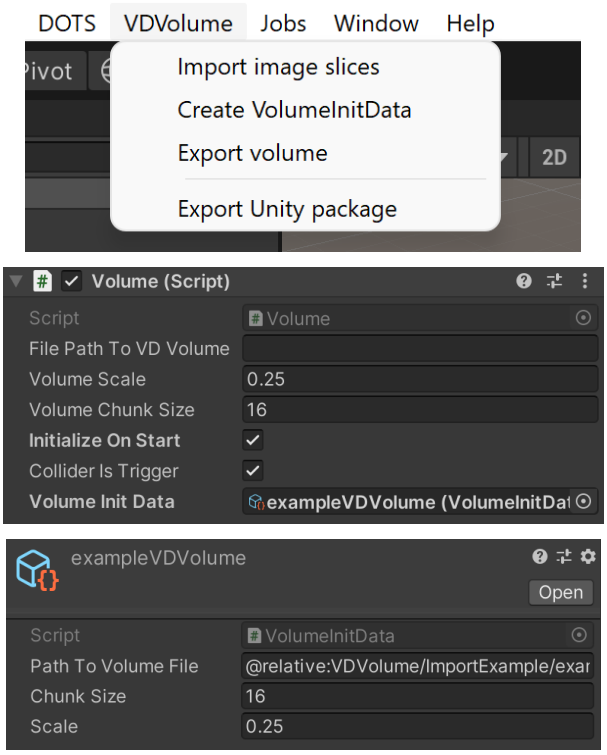

Figure 43: Various GUI elements inside the Unity editor. Top: "VDVolume" menu. Middle: Inspector of the volume script. Bottom: Inspector of the "VolumeInitData" object.

Another initialisation argument is the chunk size, which is the number of voxels a chunk measures on one axis. With the scale argument, you can scale the volume. For further scale operations, you have to call a scale method on the volume object. Scaling can not be made a continuous frame by frame operation since scaling the volume will also trigger a re-initilisation of the collider object pool in the `VolumeCollidable` script, if such a scipt is attached to an object present in the scene. This is because the number of colliders depends on the volume resolution and scale, and also on the size of the collidable object. Translations and rotations can be made on the volume object like any other unity object.

The path argument can be relative by adding `@relative:` in front of the path. A relative path starts at the "StreamingAssets" folder of your project. When loading or saving a volume, the operation will be asynchronous and executed by a .NET thread. While the operation is pending, there will be a billboard displayed in the scene (see Figure 44). This billboard will read "Loading…" or "Saving…" depending on the process and will always face the user camera.

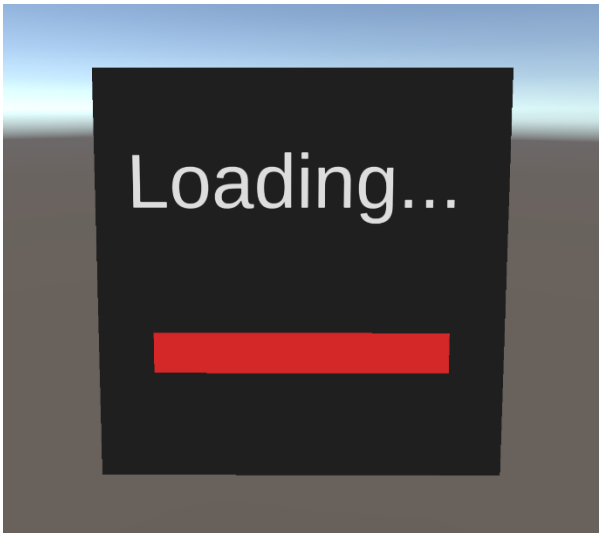

Figure 44: Billboard which shows the progress when loading or saving a volume.

6.11. Exporting a volume

With the "VDVolume" menu, it is also possible to export a `.vdvolume` file to PNG images. When clicking "Export volume" inside the menu, it will open up a new window (see Figure 45).

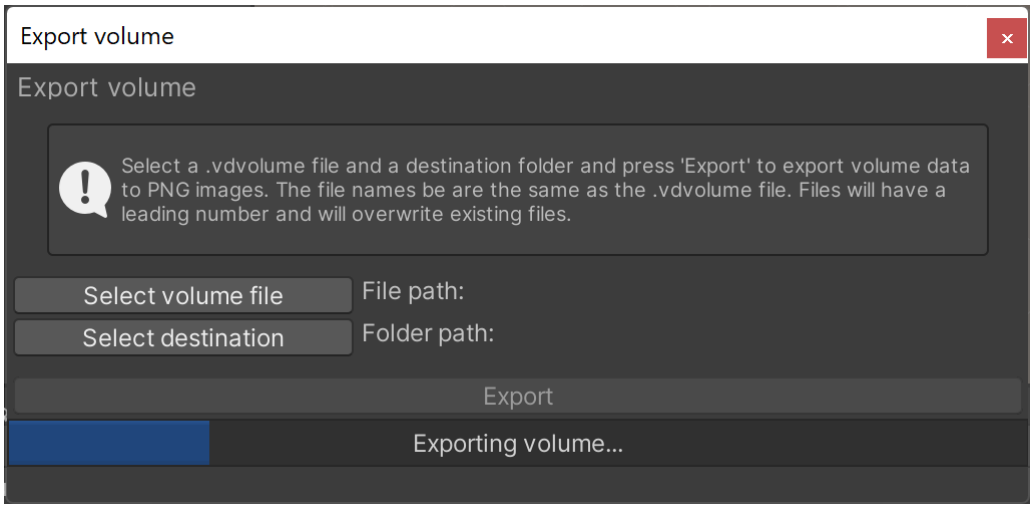

Figure 45: The Unity editor window for exporting a ".vdvolume" file to PNG images.

On this window you can specify the `.vdvolume` file and the destination folder. Once everything is selected, you can export it by pressing "Export". Exported image slices may be useful when one wants to import the volume into a different program like 3D Slicer. When exporting a volume to PNG images, removed voxels will be transparent, while solid and visible voxels will be opaque. This is accomplished through the alpha channel of the PNG file format (RGBA, 32 bits per pixel). Exporting a volume is an asynchronous operation and will be executed in a .NET thread. During the operation, a progress bar will show the progress.

6.12. User documentation and unit tests

A user documentation for the application (see Figure 46) was created with Doxygen [65]. The user guide provides the following sections:

- **Installation**:
  This section shows you how to install the library to your Unity project.
- **Getting started**:
  A simple guide for creating your first volume and cutting it with a sphere.
- **Import image slices**:
  Introduction to how to import image slices from a CT scan by creating a `.vdvolume` file.
- **Working with a volume**:
  This page will tell how to create, initialise, save and scale a volume and what events a volume provides.
- **Volume modifiers**:
  This section includes information about cutting, restoring, undoing and filtering of a volume.
- **Querying voxels and volume information**:
  Explanations of how to query a voxel and retrieve volume information.
- **Working with colliders**:
  This section explains how you can create a custom object which collides with voxels of the volume.
- **Export volume**:
  This section explains how you can export a `.vdvolume` file to PNG images.
- **Examples**:
  Code examples on how you can use VDVolume, available prefabs, example scenes and example volume data.
- A class list with descriptions of all public methods and properties.

The documentation can be generated by installing Doxygen and executing “doxygen” in the “Documentation/Src” folder of the source code.

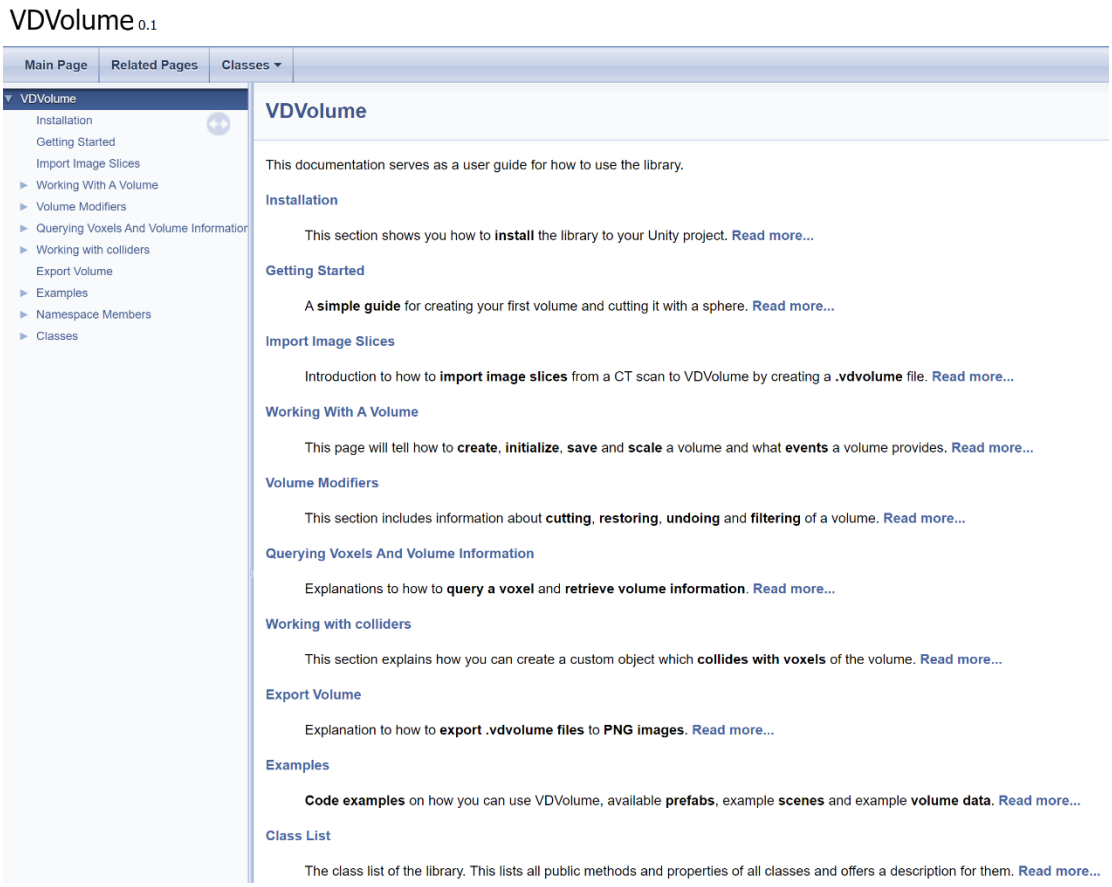

Figure 46: Doxygen user documentation of the application.

Furthermore, all relevant code is tested with unit tests and is fully commented.

7. Evaluation

Two different kinds of evaluation were performed. The first one evaluates the performance of the implementation with a benchmark providing quantitive data. The second one evaluates the solved usability problems with a heuristic evaluation providing qualitative data.

7.1. Benchmark

The benchmark evaluation has to goal to give an impression of how well the application runs for different operations and what the limitations are. For all benchmarks, the best result of 10 runs was taken. For all benchmarks except for "7.1.1. General performance compared to the other libraries", the results were taken from a development build, analysed with the profiler inside the Unity editor. The Valve Index and the haptic device were not connected for the benchmarks to get an isolated testing environment. The build of the application had the following properties:

- Resolution: 1440x1600
- Quality setting: high
- MSAA: disabled
- Default contact offset: 0.0001
- Fixed physic time step: 0.01111
- VSync: 90 fps

The resolution is the same as one display of the Valve Index. Since the scene has to be rendered twice for the headset, the rendering related data from the benchmarks should be considered more demanding in a real-world scenario. MSAA has been disabled since one may want to switch to another faster anti-aliasing solution. The physic time step was set to match the frame rate. The frame rate was limited to 90 fps to match Valve Index's 90Hz mode. The default contact offset was set to a much lower value due to the fact that the volume inside the scene was scaled to 0.003 which matched the scaling from the existing prototype where everything is scaled down due to the default settings of the SteamVR experience in Unity. The PC specifications used for benchmarking were the same as in section "4.2.3 Result". The memory usage after starting the application and initialising of the volume is 1.84 GB

according to the Unity profiler while Windows Task Manager tells 1.3 GB. Since everything is pre-allocated, only a few megabytes may be added when doing some operations.

7.1.1. General performance compared to the other libraries

Let's first compare the implementation (VDVolume) to the other libraries used in "5. Voxel library comparison". Like with all other libraries, VDVolume was integrated into the benchmarking command line tool to automatize the benchmark process. The results can be viewed in Table 8.

Operation/Library	Cubiquity	Voxel Play	OpenVDB	VDVolume
Initialise (seconds)	39.139	6.29	5.85	2.275
Delete (seconds)	0.283	0.115	0.041	0.123
Add (seconds)	0.295	0.05	0.041	0.145
Modify (seconds)	0.295	0.05	0.048	0.015

Table 8: Performance comparison of VDVolume to the other selected libraries.

As you can see, VDVolume beats Cubiquity in every category. The "Initialise" and "Modify" operation is faster than all other libraries, while the "Delete" and "Add" operation is slower than Voxel Play and OpenVDB. The deficit of the "Add" operation probably comes from the more expensive operations to maintain the visible voxels list. The "Delete" operation is only slightly slower than Voxel Play. That is because the operations to maintain the visible voxels list when deleting are simpler. The "Modify" operation is fast because it only accesses the dense storage and writes new colour values to the 3D array without the creation or removal of visible voxels. The same reason applies to the "Initialise" operation since this method primarily accesses the 3D array, followed by making the edge voxels of the volume visible, which adds voxels to the visible voxel list. In general, you could say that the performance is heavily affected by maintaining the "only-show-visible-voxels" concept. Since OpenVDB does not has such a concept for a dense grid, it is obviously way faster in this benchmark since the benchmark code only turns values on or off when deleting or adding without caring about managing the relevant visible voxels. Therefore, the "Modify" operation values of OpenVDB and VDVolume are more comparable than the others. Which also means that the values of Cubiquity, Voxel Play and VDVolume are more comparable since they incorporate a "what-to-render-efficiently" concept for dense grids when deleting, adding, initialising and modifying while OpenVDB in this benchmark does not. While making these tests, it was also tested how parallel cutting would perform to show the potential of that approach. When cutting a cube with the same size used in this comparison, the operation took 0.006 seconds, while adding a cube in parallel also took 0.006 seconds.

7.1.2. Cutting performance compared to the existing prototype

How does the actual cutting performance compare to the existing prototype? To answer this question, the exact setup and implementation used in "4.2.1 Setup" and "4.2.2

Implementation" was used. The volume and method calls were swapped out with the equivalents of VDVolume.

Ray cast method

To see how VDVolume compares to the setting where Cubiquity was not sufficient, the scale of the sphere of the drill head was set to 0.06 which removes, like before, 4,139 per cut. As you can see in graph A of Figure 47, it delivers a smooth 90 fps experience, with a lot of room for larger head drill sizes since the application mostly waits for VSync.

Collider method

When testing cutting with the collider method, the sphere had the same default scale of 0.025 as in the existing prototype, while the volume had a scale of 0.003. With this configuration, the drill removes 504 voxels per cut. You can see in Figure 47 in graph B1 that it provides a smooth 90 fps experience. In graph B2, you can see how many colliders were present at each time. At maximum there are about 1.3 thousands enabled colliders in the scene and a maximum of 811 contact points. The jitter in the graphs comes due to the fact that the physic time at the beginning did not trigger at the same time as the head drill was moved. The spike at the start of B1 comes from initialising the object pool of the collidable object. Since changing the size of the collidable object also results in a different amount of possible colliders, this has to be done when changing the size of the colidable or volume.

Parallel cutting

In graph C of Figure 47, you can see the performance when using parallel cutting. Since there is currently only support for parallel cube cutting the sphere in the scene was changed to a cube that uses ray casting for collision detection. The scale of the cube was set to 0.1 which removes 35,937 voxels per cut. Even with this high number, it provides a solid 90 fps experience. If compared to Figure 11 of "4.2.3 Result" it can also be seen that there are no random spikes in the graph. There are some parts in the graphs where rendering takes longer than usually. This corelates to the moment when the main window was switched to Unity to stop the recording and therefore aren't relevant. It can also be seen that the rendering in general takes up more time than Cubiquity, which is because Cubiquity only renders the visible voxel faces. Presumably, it also does not render occluded voxels. VDVolume in the current implementation renders cubes with 12 triangles and also renders those voxels which are occluded by other voxels. This generates huge overhead of calculations which are compensated by GPU instancing. How this can be improved can be read in chapter "8. Future improvements".

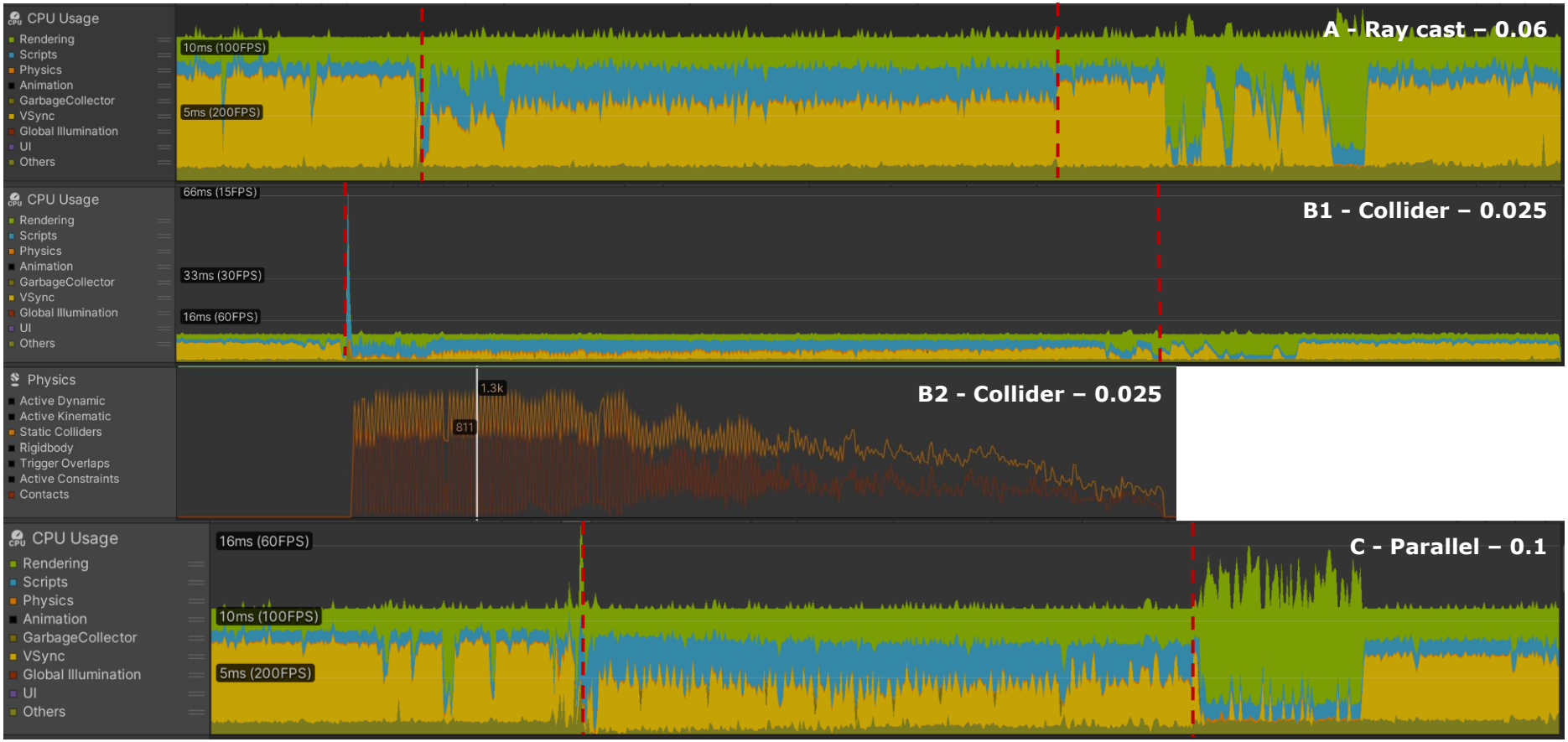

Figure 47: Cutting performance of the ray cast method, the collider method, and the parallel method. Red lines mark the start and end of each benchmark. Label in the top right corner: Image identifier – collision detection method – drill head scale.

7.1.3. Performance of undo, filtering and volume initialisation

To gather data about the performance of the undo operation, the filtering and the volume initialisation, the setup and implementation of the previous cutting benchmark was used.

Undo

After cutting the volume with a sphere of scale of 0.05 that uses the ray cast method, the undo operation was executed. The undo operation undos all voxels in the buffer. The voxel buffer size for the undo operation varied. The results can be viewed in Table 9. The undo operation in general can not be done in real time except when setting the buffer size to a small size.

Voxel buffer size	131,072	262,144	524,288	1,048,576
Time (seconds)	0.05	0.10	0.19	0.37

Table 9: Perfromance results of the undo operation.

Filtering

Two simple filter operations were executed. Before each of them, the volume was re-initialised. The results can be view in Table 10.

Value range	0-42	0-255
Time (seconds)	1.13	1.24

Table 10: Performance results of the filtering operation.

Initialisation

This test is different from the initialisation test performed in "7.1.1. General performance compared to the other libraries". This one includes rendering and reading a ".vdvolume" file, and reflects the actual user experience while the other does not. The best out of 10 values for volume initialisation is 6:59 seconds.

7.1.4. Limitations

This section aims to answer the question under what circumstances does the application stop delivering a 90 fps experience. For all cutting performance tests in this section, the setup and implementation from the previous section was used.

Ray cast method

Starting with the limitation of cutting when using the ray cast method, graph A in Figure 49 shows the performance when setting the sphere scale to 0.087, which removes 11,459 per cut. The limiting factor of this method is the iterative removal of voxels.

Collider method

Graph B1 shows cutting with the collider method. The scale of the sphere was set to 0.03, which removes 792 voxels per cut. Graph B2 shows that a maximum of about 1,9 thousand enabled colliders were used at the maximum. The limiting factor of this method is the

number of colliders, which depends on the volume and the collidable object scale. If the number of the colliders is too big, the collidable script has to iterate over too many possible colliders. Especially updating the corresponding game object of the colliders is the most demanding task. Which means if there are less enabled colliders to update, the performance will be better even when the script has to iterate over the entire AABB described in "6.7. Physic collider". This is also the reason why, in the profiler graph, the performance increases over time since the cutting object will have smaller steps and doesn't have to create colliders where voxels are already removed. The calculation of the contact points which Unity provides will also increase. However, that is just a small percentage compared to the collider iteration.

Parallel cutting

Graph C shows parallel cutting. The scale of the cube is set to 0.15, which removes 132,651 per cut. Here, the limiting factor is the number of cores available for executing the cube layers mentioned in "6.5.3 Parallel cutting and restoring". Also, the way the cube is divided into separate data for the threads is sub-optimal. The larger the cube gets, the larger is also the amount of voxels a cube layer has, increasing the execution time of a thread. The data should be divided more equally. And at some point, the rendering will limit the parallel cutting operation.

Rendering

Another limiting factor is the number of visible voxels that the application can render. Visible voxels in this context refer to the definition in "6.3 Model". It shows that the application can handle about 2,476,484 visible voxels. To come to this conclusion, the setup and implementation of the parallel cutting benchmark was used. A parallel cutting benchmark with a cube size of 0.2 has been made. After that benchmark was finished, it left the volume with 2,476,484 visible voxels. Because the spiral path the drill head uses is inside the volume, most of the visible voxels are occluded from other visible voxels. The profiler graph of this benchmark can be seen in Figure 48. There you can see that at the end of the operation, the rendering is constantly slightly below 90 fps. The rendering takes too much time so that the rendering thread of Unity has to wait for the main thread and vice versa. How this can be improved will be explained in "8. Future improvements".

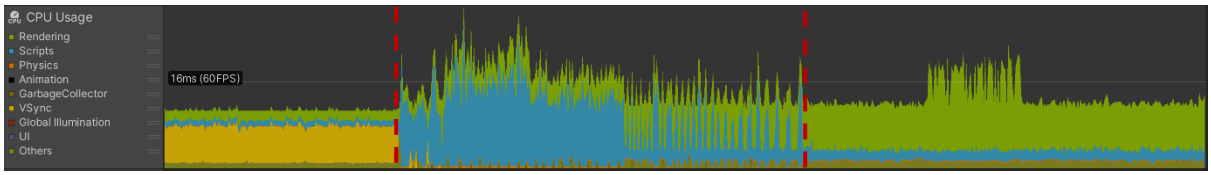

Figure 48: Performance of parallel cutting when using a cube scale of 0.2, which removes 300,763 voxels per cut. Red lines mark the start and end of the benchmark.

To put this result into perspective in Table 11, you can see how the visible voxels vary after initialising the volume and after applying a filter operation of a value range of 0-42. The volume of size 512x348x176 used for these benchmarks stores 31,358,976 voxels which are divided into 8,712 chunks that can be represented by a 33x22x12 chunk grid.

	After initialisation	After filter operation (0-42)
Visible chunks	2,344	4,197
Visible voxels	654,936	1,225,995

Table 11: Putting the rendering limitation of 2,476,484 visible voxels in perspective: The first data column shows the visible voxels after initialisation of the volume and the second data column shows the visible voxels after applying a filter operation with a value range of 0-42.

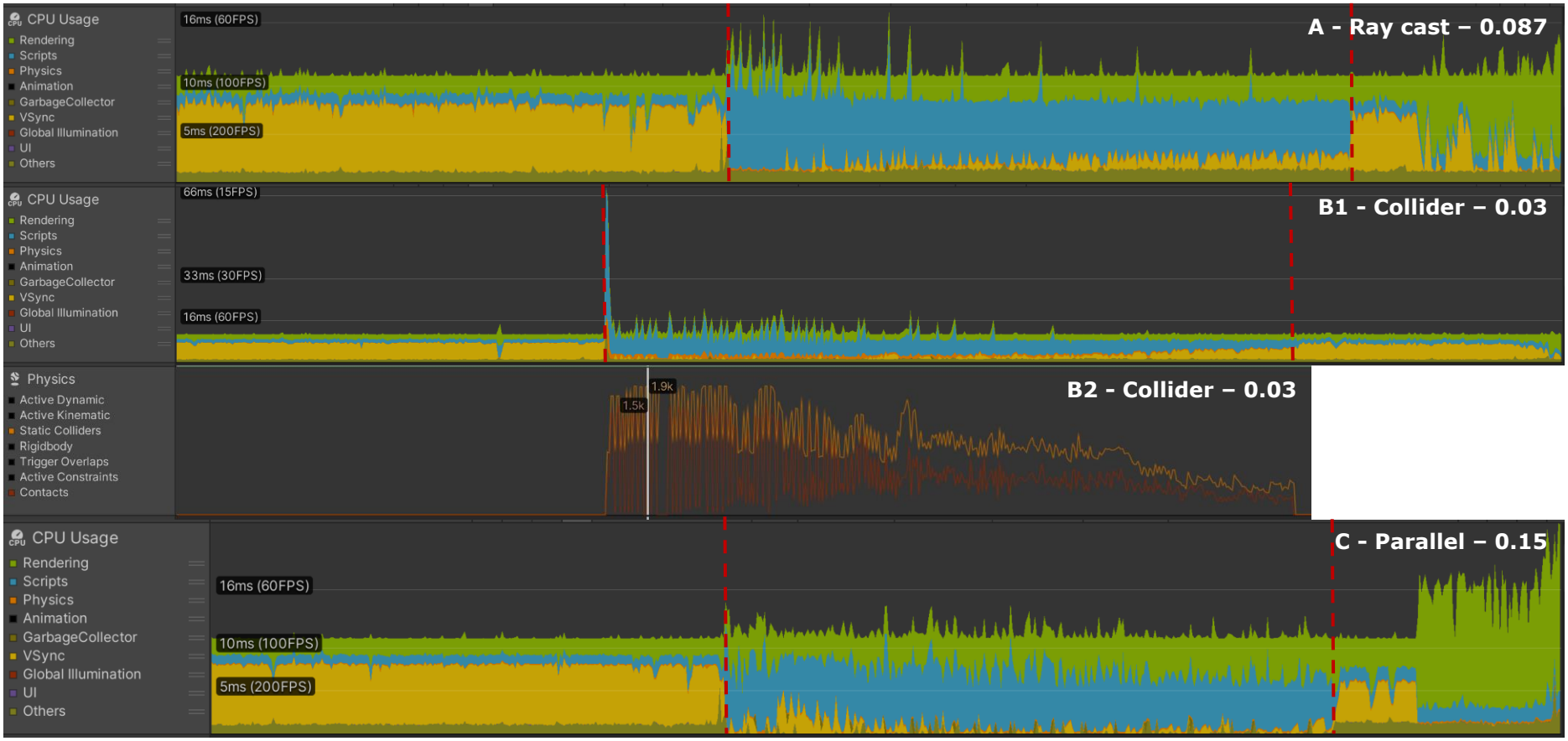

Figure 49: Performance limitation of the ray cast, the collider and the parallel cutting method. Red lines mark the start and end of each benchmark. Label in the top right corner: Image identifier – collision detection method – drill head scale.

7.1.5. Summary

From the benchmark, it is clear that the performance when cutting the volume has improved compared to Cubiquity. There are some downsides when it comes to the rendering and using the collider method. Both can be improved, which is discussed in "8. Future improvements".

The collider method can be seen as a precise method for cutting the volume. It does not support large object sizes because of the collider limitation. But when working with precision, a user may not want to work with large objects sizes since you want to work in a small area with precision. Furthermore, the performance is better when the object is not fully immersed in the volume. This is usually the case when a user cuts a volume, especially when the haptic force feedback holds the user back from putting the drill head into the volume. The cutting object is always just partially immersed in the volume, which creates less enabled colliders, increasing the performance as described in "7.1.4 Limitations". For coarse cutting method, parallel cutting can be used. When you want to work coarsely, you probably want to remove large chunks of data. For that, parallel cutting can provide fast coarse cutting.

In general, the application is not yet ready for extensive usage. The main reason of that, besides the limitations of the collider method, is the limitation of the number of voxels the application can handle. When working heavily with a volume of a size used in the benchmarks, you come easily to rendering limit described in "7.1.4 Limitations". Therefore, one has to consider implementing the points mentioned in "8. Future improvements" before moving on implementing more features.

The most relevant data from the benchmark can be viewed summarised in Table 12. There you can see that when it comes to cutting using the ray cast method, the previous prototype used, VDVolume can delete about 2.7 times more voxels than Cubiquity, while still maintaining a stable 90 fps experience.

Approx. voxel limit when cutting	792	4,139	11,459	132,651
Object scale	0.03	0.06	0.087	0.15 (cube)
Method	Collider	Ray cast	Ray cast	Parallel
Library	VDVolume	Cubiquity	VDVolume	VDVolume

Table 12: Summarised performance limits of Cubiquity and VDVolume when cutting.

7.2. Heuristic evaluation

The heuristic evaluation has the goal of verifying the solved usability problems and to gather new feedback regarding the state of the new prototype.

7.2.1. Setup

The heuristic evaluation takes place in an ordinary office room. The participants sit in front of a computer with the Valve Index on their head, the touch device on their right side and with one VR controller on their left hand (see Figure 50).

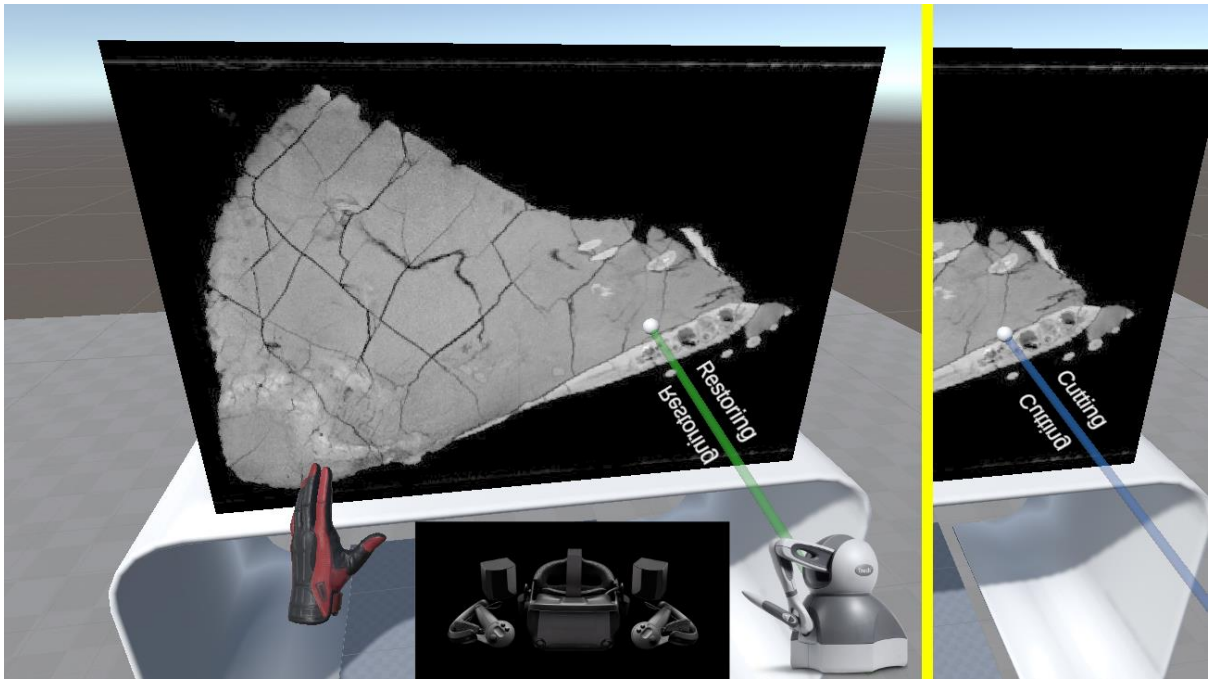

Figure 50: VR enviroment of the implemented prototype. The line attached to the sphere (drill head) serves as guide to coordinate the virtual appearance of the touch stylus inside the scene. The line is green when restoring and blue when cutting the volume.

Participants are given the task of cutting the bedrock along the border of the fossil. This imitates the common task of exposing a fossil or a part of it. Prior to starting with the task, the participants have to read the description of the heuristics the evaluation is based on. The heuristic evaluation conducted prior to this one used the 12 heuristics of Sutcliffe and Gault (see "3. Specification analysis"), but only 4 of them are related to the usability problems the prototype tries to solve. These heuristics are "Natural engagement", "Compatibility with the user's task and domain", "Close coordination of action and representation" and "Support for learning". Because of missing project time, the functionality to scale, translate and rotate a volume are not tested. A simple idea for testing those would be the ability to hold the volume in the left hand while cutting. The ability to change the drill head with different objects, like a torus or a pyramid, is also not tested. However, these ways of working are supported by the implementation. The participants have to comment on these heuristics while they are working with the volume. There are also questions by the instructor that guide the participants through the heuristics if they have no thoughts about it.

There is also a second part of the evaluation. Because of time constraints, the filtering could not have been implemented as a VR interaction. Therefore, there is an additional scene on which the participants have to comment on. The participants take off their headset and watch a debug scene on a monitor (see Figure 51).

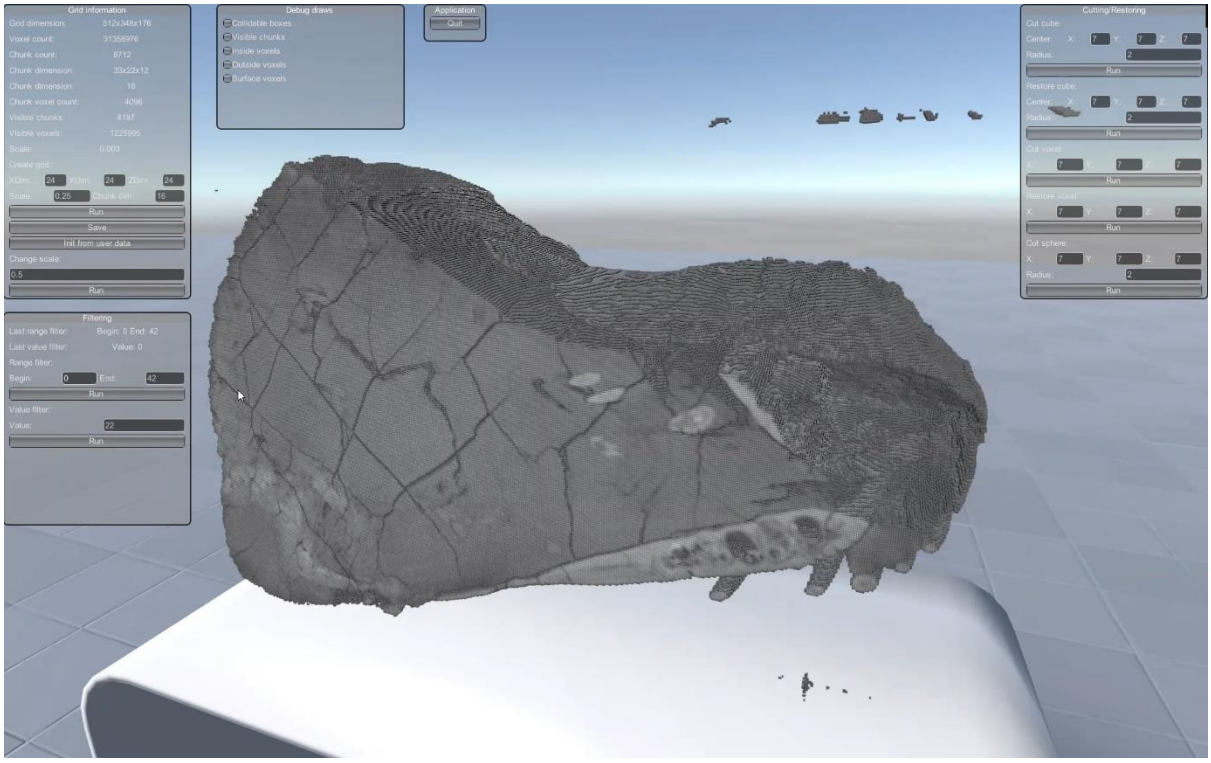

Figure 51: Debug scene of the prototype. The volume is filtered by a value range of 0 to 42.

The instructor of the evaluation will perform a filtering operation with a value range of 0 to 42 on a 2D UI. After seeing this, the participants have to answer how this functionality could be implemented in VR, in the best case without a UI. In addition to that, they also have to answer if the filtering should be a functionality provided in VR or if it is a task users should do before they import the images slices into the application.

7.2.2. Implementation of the prototype

For the heuristic evaluation, a new git branch “VDVolume” was added to the existing prototype repository. This branch replaces Cubiquity by replacing all calls to Cubiquity with equivalent calls to VDVolume. With these changes, the existing prototype works the same as before, just with VDVolume as a voxel library. In addition to that, scripts for the heuristic evaluations were added to the folder “Assets/VDVolume_Heuristic_Eval”. These scripts are used for the heuristic evaluation. The evaluation can be loaded with the “VDVolume_Heuristic_Eval” scene inside the same folder.

In general, the scene for the heuristic evaluation has not changed compared to the “Box Collider with Rigid Body” scene used in the last evaluation. However, everything is scaled up to work with simpler numbers. For example, the volume scale of the original prototype is set to 0.003, which now is set to 0.1. Everything else is scaled up accordingly to match the

perceived object sizes of the original scene. The haptic feedback algorithm stayed the same as before since there was no focus on that in this thesis, but the render frequency that was set to 90 hz like in the benchmarks. Functionality-wise, users now can cut the volume using physic colliders which provide better precision. They also can restore the volume in the same way they cut the volume. They can switch between cutting and restoring by pressing the rear button of the pen of the touch device. New is the way how users can activate cutting and restoring. Now users have to press and hold the front button to keep the functionality active. Furthermore, you can undo the last cut or restored voxels by pressing the front and rear button at the same time. You can also resize the drill head by pressing the "A" or "B" button of the left VR controller. There are three sizes: small (object scale of 0.25), medium (object scale of 0.5), and large (object scale of 0.75). The object scale values compared to the scaling of the benchmarks are as follows: 0.0075 (small), 0.015 (medium), 0.0225 (large). Note: the default sphere scale of the original prototype is set to 0.025 but did not fully remove the voxels occupied by the sphere volume (see "4.2.2. Implementation"). The "A" button decreases the size while the "B" button increases it.

7.2.3. Execution

The evaluation was performed in two sessions with 3 people. Two of them, forming the first session, are researcher at the HTW Berlin and are familiar with VR and human computer interaction. And the other one, forming the second session, is a researcher at "Matters of Activity" and familiar with paleontology. There were two instructors, one on site and one connected remotely via a video call. After a short instruction on how to use the application, the participants started with the task. There was no time constraint, but each of them finished the task and discussion within 40 minutes. After completing the task, they were grouped up, opening a discussion for any remaining thoughts regarding the four heuristics. After that has ended, the instructor on site showed them the filtering, and the participants were asked to comment on it.

7.2.4. Result

In the following paragraphs, only key comments from the participants are listed that are new (in relation to the last heuristic evaluation) and are related to the implementation or the added functionalities.

Natural engagement

- Switching between cutting and restoring with the rear button of the stylus was found to be annoying. If there are only two functionalities, the front button should only be used for cutting when holding it down, while the back button only should be used for restoring. Otherwise, add virtual tools on top of the table. Users could pick a functionality by selecting one of those tools.

- The fossil could be colourised, making it easier to distinguish parts of the fossil. When working only with grayscale values, it limits the visualisation.
- The undo functionality was often executed accidentally because the stylus buttons are too close to each other.

Compatibility with the user's task and domain

- The restoring functionality was easy to use and fulfilled its purpose. Compared to the undo functionality, the participants liked that you can point to the voxels that should be restored.
- Users didn't know what voxels will be undoed when executing the undo functionality. Usually it was too many.
- One participant thought that the drill head did remove more voxels than the sphere volume covers.
- Visual feedback when cutting is missing. When cutting, voxels could fall out of the volume and onto the ground.
- Filtering should be executed via a 3D UI. For example, one could select a value range with sliders or maybe with a color picker.
- Filtering should be kept inside the VR experience. There might be occasions where you don't know beforehand what you want to filter, as it depends on the current state of your work with the volume. User should not have to leave the application for that.
- Filtering could be made to work on selected areas.

Close coordination of action and representation

- One participant didn't like it to be inside the default Unity sky box. It is not a realistic environment and feels uncomfortable.

Support for learning

- The undo functionality should be mapped to the left controller, not to the pen. Otherwise the learning is hard. It could also be executed through an interface which is always visible on the viewport.

- Holding down the front button for cutting and restoring found positive resonance. You don’t accidentally remove voxels anymore when moving the pen away from the volume.
- The virtual hand should be swapped with a virtual controller. In addition to that, there should be cues that tell which button does what.

Other ideas

- It is hard to see which voxels are removed when cutting the bed rock because you can't distinguish the black voxels. The Phong shading does not help there. It is also not possible to see what is behind the visible voxels. This can be frustrating because you never know if you will cut voxels that shouldn’t but cut. Furthermore, you also can not see what voxels are behind the drill head. This also leads to the same frustrating experience. Idea: drill head and bed rock voxels (or all voxels) should be made transparent. The transparency gives the user more spatial information about the fossil and could be adjusted by a slider. The new workflow created from this diverges from the real world workflow when cutting a fossil where you can not look through stone.
- More information about the volume displayed on a UI would be great. For example, what colours were removed or how much voxels were removed.
- Giving the user the ability to colour voxels would be helpful for making mental notes. A mental note could be an area where the user is not sure if it is part of the stone or fossil. This way a user can abandon an area for a while and come back to it later. Functionality could also be added to these coloured voxels. For example, to remove all the coloured voxels or to make the voxel resolution of that area higher.
- Having a small preview image of the entire fossil could help to keep track of the state of the cut fossil when working on a local area of the fossil.
- Having a mirrored volume with colourised removed voxels in the scene could help to keep track of the work that has been done.
- Instead of voxels, a smooth surface generated by the marching cube algorithm could result in a better looking visualisation.
- The workflow could be inverted. Starting with a small portion of the volume, users could segment or explore the fossil by restoring voxels instead of cutting voxels.

7.2.5. Analysis

The participant had no problems cutting and restoring the volume. The same is true for the new way of initialising cutting and restoring by holding down the front button of the stylus. However, the undo functionality should be improved. The participants did not know which voxels would be undoed when executing the undo functionality. Maybe this could be improved by only undo voxels of the last cutting or restoring stroke. Or maybe there could be a way to highlight the voxels that will be undoed. Also, there should be an overhaul of the action mapping. Having most actions mapped to the stylus was not received well. Furthermore, the filtering should be implemented as suggested as a 3D UI.

Besides the usability problems, there were also a lot of new ideas. Two of them stick out and should be explored. The first one is to make the voxels transparent to give the users more spatial information about the fossil. This would improve working with the volume a lot since you could have a better idea where and how to cut the volume without removing voxels that you didn't intend to remove. Showing only the surface of the volume blocks the visibility of the user too much. The second interesting idea is to invert the workflow of the prototype. Currently, users start with the volume entirely visible to them. They try to cut away voxels to segment bones or explore the fossil. Another approach could be to start with just a portion of the volume, letting the users restore voxels in whatever direction they desire. This could turn out to be a more engaging way of working with a fossil which you simple can not do in the real world. The same is true for the first mentioned idea.

8. Future improvements

There are a lot of open issues which should be solved before continuing developing new or improving existing end-user functionalities. Those all relate to the implementation of the voxel library, the basis of the application.

Rendering

The most important one is the limitation of the rendering. To address this issue, you have to implement chunk occlusion culling, which will reduce the number of voxels to be rendered. With chunk occlusion culling, only voxels in a chunk will be rendered when the chunk is visible from the camera. To achieve this, you could implement it the same way the developers of Minecraft did. A developer talks about it on his blog [66] and also provides interactive JavaScript examples of how it works. Essentially, they are using a flood fill algorithm to obtain a chunk visibility graph and perform a broad-search algorithm with some heuristics on the chunks to decide which chunks to discard. The results can be viewed in Figure 52. The algorithm also includes early frustum culling, which the implementation is also missing. However, when working in VR, the volume is most likely entirely in the view of the camera at some times. Therefore, the rendering has to be fast enough even if the entire volume is in sight. Another solution for chunk occlusion provides Voxel Play 2. They use an octree-based culling system which is not documented or explained anywhere but could be reverse engineered from the source code.

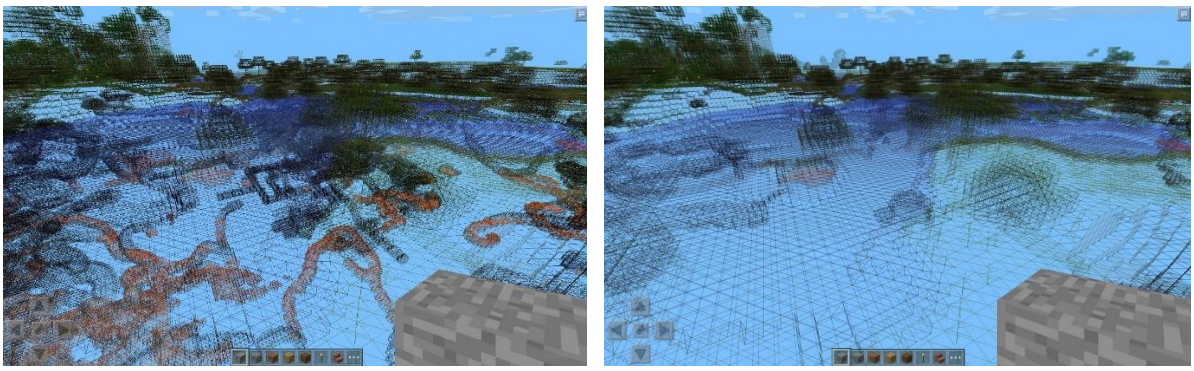

Figure 52: Result of the chunk occlusion culling algorithm Minecraft uses [67]. Left: Scene geometry without chunk culling. Right: Scene geometry after chunk culling.

The next important point is to implement greedy meshing. Currently, all 12 triangles of a voxel are rendered. However, in a worst-case scenario, only 6 are visible. With greedy meshing, you reduce the number of triangles rendered for each voxel. Greedy meshing not

only outputs just the visible triangles, which would be already an improvement, it also tries to merge triangles into one single triangle or quad (see Figure 53). Overall, it will produce less triangles to render. How this can be done is described by Lysenko in his blog [68] [69]. He also provides a JavaScript implementation [70] while a C# implementation can be found by the GitHub user Vercidium [71].

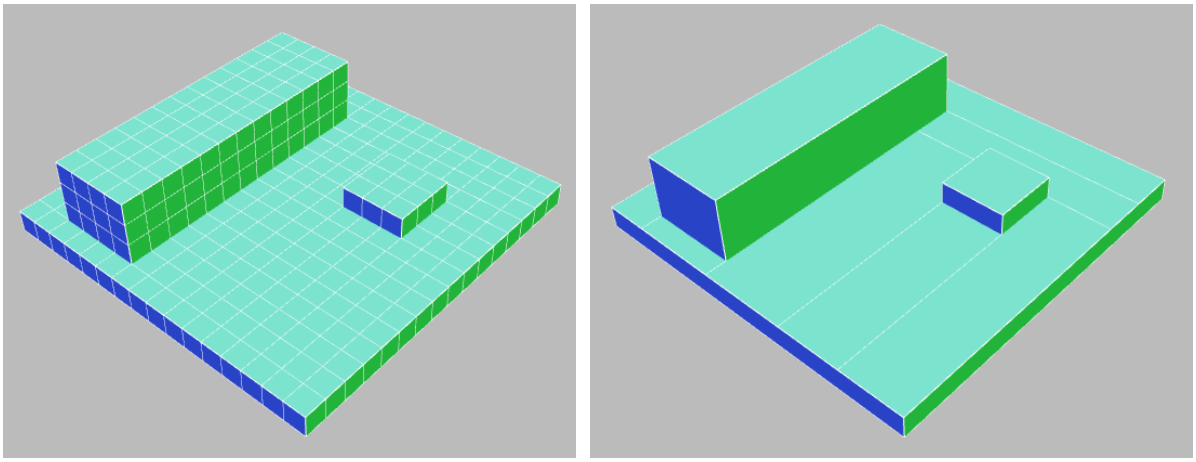

Figure 53: Result of greedy meshing [68]. Left: Voxels without greedy meshing count 690 quads. Right: Voxels with greedy meshing count 22 quads.

Collider

The next open issue to focus on is to increase to number of colliders the volume can support. This can be done in different ways. The easiest way would be to schedule the algorithm of updating the colliders in parallel with Unity DOTS. With Unity DOTS, you would not create game objects for colliders which has a huge impact on the performance when updating the colliders on mass. You would work with entities which are not part of the traditional scene hierarchy you see in the Unity editor. With entities you can also work with box colliders, and with the Job system, you would reposition and enable the colliders in parallel. Furthermore, the entities could also be made cache coherent (currently the object pool of game objects is allocated in the managed C# runtime) which would lead to more cache hits since the script iterates over the same data every frame. The AABB, which is used currently for testing for voxels, could be divided into chunks which can be assigned to jobs.

Another way, or in addition to that, could be to not check for voxels in an AABB, but to only create colliders on the object surface or to create one large collider box inside the collidable object. Both ways are difficult for non-primary objects like a sphere, cube, cylinder, etc. Another idea would be to create colliders only in the direction the pen is facing. However, this would lead to problems when moving the pen backwards while inside the volume. Also, when adding colliders only on the surface, it would not detect collisions inside the object when the object, in one time step, is suddenly inside the volume.

Ray casting

One of the last major improvements is to increase the accuracy of the ray casting (see Figure 54). You can easily achieve this with an algorithm which is based on DDA (Digital Differential Analysis). Vandevenne [72] describes such an algorithm which works fast and robust for a tiled world like a voxel grid. There is also a C++ implementation of it by GitHub user OneLoneCoder [73]. In the comments of the source code, he also linked a YouTube video where he explains the algorithm step by step.

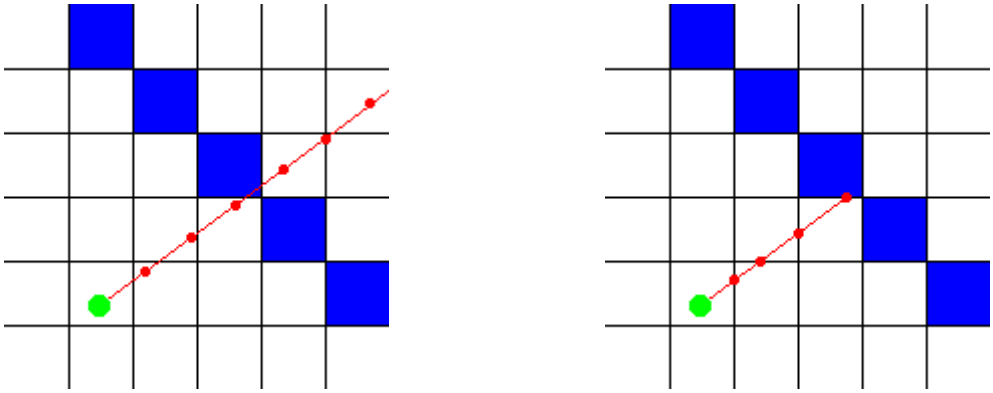

Figure 54: On the left you can see the current implementation of the ray casting. On the right you can see the result when using an algorithm based on DDA [72].

Other improvements

There are also some minor improvements for the applications one could think of. For instance, more object forms for parallel cutting could be added. The way the data is divided for the threads can also be improved. Another improvement could be the support for adding custom the data to the grid. This can be done in a different way already by defining your own 3D array grid with custom data like Boolean values or float vectors outside of the application. A support for the application would be beneficial since you could also save and load the custom data. Having custom data could help to work out an algorithm for the haptic feedback which may have to rely on different data other than the colour values of the voxels. There is also the potential to further improve the ".vdvolume" file by compressing the file with Huffman coding. This would reduce the file size, making it easier to share.

9. Summary and outlook

It has been shown by benchmarks that VDVolume is faster and more robust than Cubiquity, improving the performance of the prototype. In "7.1.1. General performance compared to the other libraries" it was shown that the general performance is at least 2 times faster than Cubiquity when it comes to the pure code operations "initialise", "modify", "add" and "remove". In "7.1.5. Summary" it was shown that VDVolume can remove 2.7 more voxels when removing the voxels in a Unity scene with the ray cast method. VDVolume also never produces performance spikes inside the Unity profiler graph like Cubiquity does. Despite the performance improvements, VDVolume has still a lot of potential to be further improved performance-wise. These improvements are recommended before developing new end-user functionalities and can be read in "8. Future improvements".

The resulting prototype now renders haptic feedback at 90 hz (50 hz before), supports physic colliders for more precision when cutting or restoring, supports collision with custom meshes and has three new functionalities: filtering, undoing and restoring. Through the heuristic evaluation (see "7.2. Heuristic evaluation") it was also verified that these functionalities are working fine with the exception of the filtering, which has yet to be fully implemented as a VR interaction. The new way of initialising cutting and restoring works great according to the participants. However, the action mapping to the VR controller and the stylus should be changed in the future. Additional UI may also be added to support these changes.

Thus, the prototype has been improved by performance and functionality. All relevant code is annotated and tested with unit tests. There is also a Doxygen user documentation for programmers who want to use the library.

VDVolume is a voxel library designed to fulfill one purpose: cutting and restoring a volume made of grayscale image slices. The advantage of this approach is that relatively little amount of source code is needed and that the source code is not bloated with functionalities to support every possible workflow. Therefore, it can be easily further developed in the current state. If one wants to implement direct volume rendering, for example, you would remove the sparse data storage part or let it stay depending on your intentions, and write a new rendering function inside the rendering part. Everything else can stay. It seems that the concept of just showing visible voxels works pretty well and the used technologies (Unity DOTS and GPU instancing) are also a good fit and the right choice when using Unity.

For future development, it might be useful to decouple the haptic feedback loop from Unity, and use direct volume rendering instead of a polygon-based representation of the fossil. This would allow for better haptic response and support for transparency of the volume. Faludi et al. [23] achieved a solid result with this approach by rendering the haptic feedback at 2 kHz and the visuals of the virtual reality at 90 fps.

References

[1] M. o. Activity, "Virtual Dissection," [Online]. Available: https://www.matters-of-activity.de/en/posts/2044/virtual-dissection. [Accessed 12 06 2021].

[2] 3D Systems, "Touch device," [Online]. Available: https://de.3dsystems.com/haptics-devices/touch. [Accessed 30 12 2021].

[3] S. Community, "3D Slicer," [Online]. Available: https://www.slicer.org/. [Accessed 07 03 2022].

[4] A. Fedorov, R. Beichel, J. Kalpathy-Cramer, J. Finet, J.-C. Fillion-Robin, S. Pujol, C. Bauer, D. Jennings, F. Fennessy and M. Sonka, "3D Slicer as an image computing platform for the Quantitative Imaging Network," *Magnetic resonance imaging,* vol. 30, no. 9, pp. 1323-1341, 2012.

[5] M. Sutton, I. Rahman and R. Garwood, "Virtual paleontology—an overview," *The Paleontological Society Papers,* vol. 22, pp. 1-20, 2016.

[6] I. A. Rahman and S. Y. Smith, "Virtual paleontology: computer-aided analysis of fossil form and function," *Journal of Paleontology,* vol. 88, no. 4, pp. 633-635, 2014.

[7] R. L. Abel, C. R. Laurini and M. Richter, "A palaeobiologist's guide to 'virtual'micro-CT preparation," *Palaeontologia Electronica,* vol. 15, no. 2, pp. 1-16, 2012.

[8] T. Akenine-Moller, E. Haines and N. Hoffman, Real-time rendering, AK Peters/crc Press, 2019.

[9] Umbra Software, "Next Generation Occlusion Culling," 06 03 2012. [Online]. Available: https://www.gamedeveloper.com/programming/sponsored-feature-next-generation-occlusion-culling. [Accessed 09 12 2021].

[10] Unity Technologies, "Occlusion culling," 03 12 2021. [Online]. Available: https://docs.unity3d.com/Manual/OcclusionCulling.html. [Accessed 09 12 2021].

[11] M. Stiftinger, "Der Octree," 15 10 1994. [Online]. Available: https://www.iue.tuwien.ac.at/phd/stippel/node51.html. [Accessed 06 12 2021].

[12] S. Laine and T. Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation," *NVIDIA Corporation,* 2010.

[13] E.-A. Karabassi, G. Papaioannou and T. Theoharis, "A fast depth-buffer-based voxelization algorithm," *Journal of graphics tools,* vol. 4, no. 4, pp. 5-10, 1999.

[14] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama and D. Weiskopf, Real-time volume graphics, A K Peters, Ltd., 2006.

[15] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *ACM siggraph computer graphics,* vol. 21, no. 4, pp. 163-169, 1987.

[16] H. Tariq and A. Burney, "Brain MRI literature review for interdisciplinary studies," *Journal of Biomedical Graphics and Computing,* vol. 4, no. 4, pp. 41-53, 2014.

[17] Valve Corporation, "SteamVR," [Online]. Available: https://www.steamvr.com/. [Accessed 07 03 2022].

[18] 3D Systems, "3D Systems Openhaptics® Unity Plugin," [Online]. Available: https://assetstore.unity.com/packages/tools/integration/3d-systems-openhaptics-unity-plugin-134024. [Accessed 07 03 2022].

[19] D. Williams, "Cubiquity Voxel Engine," [Online]. Available: https://bitbucket.org/volumesoffun/cubiquity/src/master/. [Accessed 09 12 2021].

[20] Valve, "Valve Index," [Online]. Available: https://www.valvesoftware.com/de/index. [Accessed 30 12 2021].

[21] H. Mallison, "Digitizing methods for paleontology: applications, benefits and limitations," in *Computational paleontology*, Springer, 2011, pp. 7-43.

[22] S. G. Izard, J. A. J. Mendez, P. R. Palomera and F. J. Garcia-Penalvo, "Applications of virtual and augmented reality in biomedical imaging," *Journal of medical systems,* vol. 43, no. 4, pp. 1-5, 2019.

[23] B. Faludi, E. I. Zoller, N. Gerig, A. Zam, G. Rauter and P. C. Cattin, "Direct visual and haptic volume rendering of medical data sets for an immersive exploration in virtual reality," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, 2019, pp. 29-37.

[24] F. King, J. Jayender, S. K. Bhagavatula, P. B. Shyn, S. Pieper, T. Kapur, A. Lasso and G. Fichtinger, "An immersive virtual reality environment for diagnostic imaging," *Journal of Medical Robotics Research,* vol. 1, no. 1, 2016.

[25] G. Wheeler, S. Deng, N. Toussaint, K. Pushparajah, J. A. Schnabel, J. M. Simpson and A. Gomez, "Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity," *Healthcare technology letters,* vol. 5, no. 5, pp. 148-153, 2018.

[26] S. Rizzi, "Volume-based graphics and haptics rendering algorithms for immersive surgical simulation," University of Illinois at Chicago, 2013.

[27] ImmersiveTouch, Inc., "ImmersiveTouch," [Online]. Available: https://www.immersivetouch.com. [Accessed 04 01 2022].

[28] S. Reddivari and J. Smith, "VRvisu++: A Tool for Virtual Reality-Based Visualization of MRI Images," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, IEEE, 2020, pp. 1129-1130.

[29] Specto, "SpectoVR," [Online]. Available: https://www.diffuse.ch/. [Accessed 04 01 2022].

[30] CHAI3D, "CHAI3D," [Online]. Available: https://www.chai3d.org/. [Accessed 04 01 2022].

[31] J. Zhang, Y. Lyu, Y. Wang, Y. Nie, X. Yang, J. Zhang and J. Chang, "Development of laparoscopic cholecystectomy simulator based on unity game engine," in *Proceedings of the 15th ACM SIGGRAPH European Conference on Visual Media Production*, 2018, pp. 1-9.

[32] G. Wheeler, S. Deng, N. Toussaint, K. Pushparajah, J. A. Schnabel, T. Peters, J. M. Simpson and A. Gomez, "Unity and VTK for VR medical image analysis--an initial clinical evaluation," *IPCAI, Rennes, France,* 2019.

[33] P. Kalshetti, P. Rahangdale, D. Jangra, M. Bundele and C. Chattopadhyay, "Antara: An interactive 3D volume rendering and visualization framework," *arXiv preprint arXiv:1812.04233,* 2018.

[34] S. You, L. Hong, M. Wan, K. Junyaprasert, A. Kaufman, S. Muraki, Y. Zhou, M. Wax and Z. Liang, "Interactive volume rendering for virtual colonoscopy," in *Proceedings. Visualization'97 (Cat. No. 97CB36155)*, IEEE, 1997, pp. 433-436.

[35] M. B. Hoffensetz and C. N. Daugbjerg, "Volume visulization of medical scans in virtual reality," 2018.

[36] D. Escobar-Castillejos, J. Noguez, R. A. Cardenas-Ovando, L. Neri, A. Gonzalez-Nucamendi and V. Robledo-Rella, "Using Game Engines for Visuo-Haptic Learning Simulations," *Applied Sciences,* vol. 10, no. 13, 2020.

[37] A. Sutcliffe and B. Gault, "Heuristic evaluation of virtual reality applications," *Interacting with computers,* vol. 16, no. 4, pp. 831-849, 2004.

[38] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," *Proceedings of the SIGCHI conference on Human factors in computing systems,* pp. 249-256, 1990.

[39] R. Murtza, S. Monroe and R. J. Youmans, "Heuristic evaluation for virtual reality systems," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting,* vol. 61, no. 1, pp. 2067-2071, 2017.

[40] J. L. Gabbard, D. Hix and J. E. Swan, "User-centered design and evaluation of virtual environments," *IEEE computer Graphics and Applications,* vol. 19, no. 6, pp. 51-59, 1999.

[41] D. Williams, "About Us," [Online]. Available: http://www.volumesoffun.com/about-us/index.html. [Accessed 09 12 2021].

[42] forum.unity.com/members/ootz0rz.239402/, "Cubiquity - A fast and powerful voxel plugin for Unity3D," [Online]. Available: https://forum.unity.com/threads/cubiquity-a-fast-and-powerful-voxel-plugin-for-unity3d.184599/page-16. [Accessed 06 12 2021].

[43] D. Williams, "Reflections on Cubiquity and finding the path forward," [Online]. Available: http://www.volumesoffun.com/reflections-on-cubiquity-and-finding-the-path-forward/index.html. [Accessed 09 12 2021].

[44] D. Williams, "Cubiquity Voxel Engine," [Online]. Available: https://github.com/DavidWilliams81/cubiquity. [Accessed 09 12 2021].

[45] ImageMagick Studio LLC, "ImageMagick," [Online]. Available: https://imagemagick.org. [Accessed 07 03 2022].

[46] SQLite, "SQLite," [Online]. Available: https://www.sqlite.org/index.html. [Accessed 09 12 2021].

[47] Academy Software Foundation (ASWF), "What does "VDB" stand for?," [Online]. Available: https://academysoftwarefoundation.github.io/openvdb/faq.html#sMeaningOfVDB. [Accessed 09 12 2021].

[48] D. Williams, "Cubiquity For Unity3D," [Online]. Available: https://bitbucket.org/volumesoffun/cubiquity-for-unity3d/src/master/. [Accessed 09 12 2021].

[49] D. Williams, "Cubiquity - A fast and powerful voxel plugin for Unity3D," [Online]. Available: https://forum.unity.com/threads/cubiquity-a-fast-and-powerful-voxel-plugin-for-unity3d.184599/. [Accessed 09 12 2021].

[50] Microsoft, "Platform Invoke (P/Invoke)," 15 09 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke. [Accessed 09 12 2021].

[51] J. Köller, "mathematische-basteleien.de," [Online]. Available: http://www.mathematische-basteleien.de/spiral.htm. [Accessed 22 11 2021].

[52] A. M. Cohill, D. M. Gilfoil and J. V. Pilitsis, "A methodology for evaluating application software," *Proceedings of the Human Factors Society Annual Meeting,* vol. 30, no. 1, pp. 14-18, 1986.

[53] V. Careil, "Voxel plugin," [Online]. Available: https://voxelplugin.com/. [Accessed 09 03 2022].

[54] Kronnect Technologies, "Voxel Play 2," [Online]. Available: https://assetstore.unity.com/packages/tools/game-toolkits/voxel-play-2-201234. [Accessed 12 12 2021].

[55] Kronnect Technologies, "About Us," [Online]. Available: https://kronnect.com/about-us. [Accessed 12 12 2021].

[56] Kronnect Technologies, "Voxel Play Documentation," [Online]. Available: https://kronnect.freshdesk.com/support/solutions/42000007642. [Accessed 12 12 2021].

[57] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM transactions on graphics (TOG),* vol. 32, no. 3, pp. 1-12, 2013.

[58] R. Hoetzlein, "GVDB: Raytracing sparse voxel database structures on the GPU," *Proceedings of High Performance Graphics,* pp. 109-117, 2016.

[59] R. Hoetzlein, "GVDB at SIGGRAPH 2016: Technical Presentation," 2016. [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/designworks/docs/GVDB_TechnicalTalk_Siggraph2016.pdf. [Accessed 05 12 2021].

[60] N. Blog, "TSDF (Truncated signed distance field)," 03 08 2020. [Online]. Available: https://m.blog.naver.com/jws2218/222049854706. [Accessed 10 03 2022].

[61] openvdb.org, "Overview of toolset," [Online]. Available: https://artifacts.aswf.io/io/aswf/openvdb/openvdb_toolset_2013/1.0.0/openvdb_toolset_2013-1.0.0.pdf. [Accessed 2021 12 2021].

[62] Kronnect Technologies, "Can I use voxels of smaller size?," 16 04 2020. [Online]. Available: https://kronnect.freshdesk.com/support/solutions/articles/42000072749-can-i-use-voxels-of-smaller-size-. [Accessed 12 12 2021].

[63] K. Museth, "Understanding Ray marching in OpenVDB," 30 01 2017. [Online]. Available: https://groups.google.com/g/openvdb-forum/c/Bl0ip1_rX2M. [Accessed 12 06 2021].

[64] J. de Vries, "LearnOpenGL - Transformations," [Online]. Available: https://learnopengl.com/Getting-started/Transformations. [Accessed 21 02 2022].

[65] doxygen, "doxygen," [Online]. Available: https://www.doxygen.nl/.

[66] T. Checchi, "The Advanced Cave Culling Algorithm™, or, making Minecraft faster," 31 08 2014. [Online]. Available: https://tomcc.github.io/2014/08/31/visibility-1.html. [Accessed 04 03 2022].

[67] T. Checchi, "The Advanced Cave Culling Algorithm™ Part 2 - Traversing the graph," 31 08 2014. [Online]. Available: https://tomcc.github.io/2014/08/31/visibility-2.html. [Accessed 04 03 2022].

[68] M. Lysenko, "Meshing in a Minecraft Game," 30 06 2012. [Online]. Available: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. [Accessed 04 03 2022].

[69] M. Lysenko, "Meshing in a Minecraft Game (Part 2)," 07 07 2012. [Online]. Available: https://0fps.net/2012/07/07/meshing-minecraft-part-2/. [Accessed 04 03 2022].

[70] M. Lysenko, "mikolalysenko.github.com - greedy.js," 06 07 2012. [Online]. Available: https://github.com/mikolalysenko/mikolalysenko.github.com/blob/gh-pages/MinecraftMeshes/js/greedy.js. [Accessed 04 03 2022].

[71] Vercidium, "greedyvoxelmeshing," [Online]. Available: https://gist.github.com/Vercidium/a3002bd083cce2bc854c9ff8f0118d33. [Accessed 04 03 2022].

[72] L. Vandevenne, "Lode's Computer Graphics Tutorial - Raycasting," 2020. [Online]. Available: https://lodev.org/cgtutor/raycasting.html. [Accessed 05 03 2022].

[73] OneLoneCoder, "olcPixelGameEngine/OneLoneCoder_PGE_RayCastDDA.cpp," 28 02 2021. [Online]. Available: https://github.com/OneLoneCoder/olcPixelGameEngine/blob/master/Videos/OneLoneCoder_PGE_RayCastDDA.cpp. [Accessed 05 03 2022].

[74] A. W. Brown and K. C. Wallnau, "A framework for evaluating software technology," *IEEE software,* vol. 13, no. 5, pp. 39-49, 1996.

[75] A. S. Jadhav and R. M. Sonar, "Evaluating and selecting software packages: A review," *Information and software technology,* vol. 51, no. 3, pp. 555-563, 2009.

[76] github.com/karasusan, "OpenVDBForUnity," [Online]. Available: https://github.com/karasusan/OpenVDBForUnity. [Accessed 05 12 2021].

[77] G. Boloix and P. N. Robillard, "A software system evaluation framework," *Computer,* vol. 28, no. 12, pp. 17-26, 1995.

[78] G. Gediga, K.-C. Hamborg and I. Düntsch, "Evaluation of software systems," *Encyclopedia of computer science and technology,* vol. 45, no. 30, pp. 127-153, 2002.

[79] International Organization for Standardization, "Software engineering - Product quality," 12 1991. [Online]. Available: https://www.iso.org/standard/16722.html. [Accessed 13 12 2021].

List of Figures

List of Tables

Abbreviations

AABB	Axis aligned bounding box
AHP	Analytical hierarchy process
AI	Artificial intelligence
CLI	Command-line interface
CPU	Central processing unit
CT	Computed tomography
CUDA	Compute unified device architecture
DDA	Digital differential analysis
DLL	Dynamic-link library
DOTS	Data-oriented technology stack
fps	Frames per second
GPU	Graphics processing unit
I/O	Input/Output
ISO	International organization for standardization
LOD	Level of detail
MRI	Magnetic resonance imaging
OpenCL	Open computing language
OpenGL	Open graphics library
SVO	Sparse voxel octree
VR	Virtual reality
VSync	Vertical synchronization
VTK	Visualization toolkit
WAS	Weighted average sum

A. Contents of the source code

- **BenchmarkCppVoxelLibraries**:
  Benchmarking CLI for automated benchmarking of Cubiquity, Voxel Play, OpenVDB, VDVolume, written in C++. Supported benchmarks: initialise, add, remove and modify. To build this program, you need to download and build OpenVDB. You also need to build the "BenchmarkVoxelPlay" and "VDVolume" project and specify the paths of the executables in the code if you want to execute these benchmarks. Only works on windows since it has dependencies to windows libraries.
- **BenchmarkCubiquity**:
  Unity application for benchmarking Cubiquity.
- **BenchmarkVDOpenVDB**:
  Unity application that renders a volume with OpenVDB. This project includes the DLL build with the "VDOpenVDB" project.
- **BenchmarkVoxelPlay**:
  Unity application for benchmarking Voxel Play.
- **VDOpenVDB**:
  Native Unity plugin to access OpenVDB. The project is written in C++ and can be used to build a DLL, which can be included by a Unity application. You need to download and build OpenVDB in order to build this project.
- **VDVolume**:
  Voxel library for Unity to display and modify a volume from a CT scan.
- **VDVoxels**:
  The "VDVolume" branch of the overarching project prototype. Essentially, Cubiquity is replaced with VDVolume. It also contains the code used for the conducted heuristic evaluation.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum: 07.04.2022 Unterschrift: