Visual Tools for Computed Tomography Volume Representation: Large Data Visualisation and Surface Extraction

Auteur : Greffe, Roland
Promoteur(s) : Geuzaine, Christophe

Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20388

UNIVERSITY OF LIÈGE

FACULTY OF APPLIED SCIENCES

Master thesis submitted for the degree of MSc in Computer Science and Engineering

Visual Tools for Computed Tomography Volume Representation: Large Data Visualisation and Surface Extraction

Author	Roland GREFFE
Supervisors	Prof. Christophe GEUZAINE Dr. Vincent LIBERTIAUX
Jury Members	Prof. Laurent MATHY Prof. Eric BÉCHET Prof. Christophe PHILLIPS Prof. Christophe GEUZAINE Dr. Vincent LIBERTIAUX

Academic Year 2023-2024

Abstract

A very important technique in the field of Non-Destructive Testing (NDT) is X-Ray Computed Tomography (CT). This technique allows the reconstruction of an inspected part in a 3D voxel volume, which can be used to detect internal defects in industrial parts. One issue of this technique is the size of the data, which can be too large to fit into the memory of the graphics processing unit (GPU), making the rendering of this part impossible.

The scope of this thesis is to develop a real-time out-of-core rendering solution which allows the rendering of CT volumes larger than the available memory of the GPU (VRAM). On top of this solution, rendering techniques specific to X-Ray CT must be implemented. These rendering modes are the ones implemented by X-Ray Imaging Solutions (X-RIS), the industrial promoter of this thesis, in their Maestro software.

Our solution is based on the GigaVoxel library, which is itself based on the paper "GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering". This library allows the rendering of large voxel volumes in real time by using a combination of level of detail techniques, occlusion culling and a page table system on the GPU. It was modified during this thesis in order to optimize the pre-processing step of voxel volumes, improve its rendering performance, and fix bugs. The rendering modes of the Maestro software were then implemented on top of it, and the initial goal of the thesis has thus been achieved. Based on our results, importation of the library into Maestro can be launched. A future work can focus on optimizing the memory usage of the library, as it works under the assumption that the whole volume always fits in the RAM.

In a second part of the thesis, different surface extraction techniques are studied. These techniques allow the extraction of a non-voxel representation of the surface of an object from a voxel representation of it, which is useful in the metrology field to measure the features of a scanned object. Unfortunately, the tested methods did not offer a good enough precision for what is required in the field of metrology, especially on the sharp edges of the tested volume. Future work shall focus on the use of a method that better deals with sharp edges and on a better way to filter out the noise in the volume caused by the reconstruction artifacts of CT scans. These changes can help reduce the error in the range of a tenth of a voxel, which is required for precise metrology applications.

Acknowledgments

I wanted to begin this thesis by expressing my greatest gratitude towards Vincent Libertiaux and Prof. Christophe Geuzaine for their help and guidance throughout my work on this thesis and our (close to) weekly meetings. And to X-RIS for allowing me to work in their offices and use their RTX 3090 for as long as I did.

I will now thank:

- My father Christophe and my brother Nathan for their help with the redaction of this thesis.
- My mother Véronique for buying me a tremendous amount of Magnums.
- My sister Fanny for her hilarious sense of humor.
- All my other family members for their support.
- The ”Quinzaine” for all of our outings.
- My friend Mathieu and my brother Martin for forcing me to keep myself (at least a bit) in shape.
- All my gaming friends (FSEI, Tristalik18, Nyde28) and the people that filled our Counter-Strike games (Elias, Lucas, Martin, Antoine, Michaël, Maxime, and the random peoples that couldn't speak English) for their help in fuelling my gaming addiction.

Contents

Introduction

Context

X-Ray Imaging Solutions (X-RIS), the industrial promoter of this thesis, is active in the field of Non-Destructive Testing (NDT). A very important technique in NDT is X-ray Computed Tomography (CT). Thanks to matter penetration properties of X-rays and reconstruction algorithms, CT provides the structure of an inspected part in a 3D voxel volume. Each voxel value in that volume is an image of the density of that part at the location of the voxel. As a consequence, this technique is a very good tool to detect internal defects (voids, inclusions, poor resin curing, . . .) or in metrology applications (internal dimensions, quality control, . . .).

One issue of CT, and of voxel volumes in general, is the size of the data. And, right now, the Maestro CT software developed by X-RIS which is used to render and analyze CT scans requires all the data to be loaded into the memory of the graphics processing unit (GPU) in order to be rendered. But the amount of memory available on a GPU (VRAM) is limited. Consumer grade GPUs have VRAM in the order of a few gigabytes (usually 4 to 8 GB) and the NVIDIA RTX 4090, which is the best consumer grade GPU available at the time of writing, has 24 GB of VRAM. This amount is not enough when higher size volumes need to be rendered. For example, a CT scan composed of 3096^3 voxels requires approximately 55 GB of storage, which is more than the VRAM of the best consumer grade GPU available at the time of writing.

The issue of rendering large volumes in real time is known as "out-of-core" rendering. It is the process of rendering data that is too large to fit into the VRAM of the GPU. There are two main families of out-of-core rendering techniques: level of detail (LOD) techniques and occlusion culling.

LOD techniques generate multiple representations of the same volume at different levels of details. Each level of detail is a lower resolution version of the volume that requires less memory to be stored. The idea is then, depending on the distance at which the volume is visualized and the amount of available VRAM, render the volume at the level of detail that is the most appropriate. Indeed, if the volume is observed from far away, it is not necessary to render it at full resolution since the user would not be able to see the loss of details. The advantage of LOD techniques is that they can work for any amount of available VRAM, as long as a small enough level of detail volume is generated. The drawback is that there is a loss of details in the volume and that a pre-processing step is required to generate the different levels of details. Since X-RIS is interested in analyzing

CT volumes, the loss of details is not acceptable.

Occlusion culling, on the other hand, is the process of determining what parts of the volume are visible to the user and only render those parts. This can be done for both parts of the volume that are outside the screen, but also for parts of the volume that are hidden by other ones of its parts. The main difficulty with occlusion culling is to implement a process that can determine what parts of the volume are visible to the user without adding too much processing time. The advantage of occlusion culling compared to LOD techniques is that its preprocessing step only requires splitting the volume into smaller parts, which is much faster than generating multiple levels of details. Occlusion culling also allows for the rendering of the full resolution volume, whereas LOD techniques may render the volume with a loss of details. The drawback is that if too much of the volume is visible to the user at once, the amount of memory required to store the visible parts of the volume can be too large to fit into the VRAM of the GPU. In practice, the best solution is to use a combination of both techniques to allow for the rendering of large volumes in real time. Indeed, when both techniques are used at the same time, the levels of detail techniques can be used to render the best level of detail volume possible given the amount of available VRAM and the occlusion culling allows reducing the amount of VRAM used by only storing the visible parts of the volume.

Plan of the thesis

The aim of this thesis was to identify, develop, and test solutions to the out-of-core rendering problem in CT. We started by looking at the available literature on both out-of-core rendering techniques for voxel volumes and volumetric rendering techniques for CT volumes. This review of the literature is presented in Chapter 1. For out-of-core rendering techniques, a study of the available data structures used to store and easily allow the implementation of both levels of detail and occlusion culling was done. Not all the techniques presented in the literature are aimed at the domain of tomography. Indeed, some were aimed at the domain of CT volumes [1], some at the domain of video game rendering [2, 3], some at the domain of microscopic visualization [4], and some for general out-of-core voxel rendering [5, 6].

In Chapter 2, the GigaVoxel library will be presented. It is an already existing solution based on the paper of Crassin et al. [5], which stands out from our review of the literature as the most appropriate tool to build our solution onto. Indeed, multiple works in the literature are based on this library and show promising results. Since the GigaVoxel library has a BSD 3 free software license, it allows developers to build on top of it and redistribute it as long as the initial copyright notice is included. This is why we decided to use it as the base of our work. The improvements made to the library will also be presented in this chapter.

In Chapter 3, the implementation of the port of volumetric rendering techniques for CT volumes that are available in the Maestro CT software will be presented. These techniques are the ones expected for CT analysis software and are the ones used by X-RIS. This chapter will question if the out-of-core methods used by the library allow the implementation of these rendering modes or not. The performance impact of these implementations will also be studied in this chapter.

In Chapter 4, an analysis of the impact of parameters of the out-of-core rendering solution on a multitude of parameters will be made. This analysis will be done on two different datasets with multiple volumes of different sizes. An analysis of the impact of the ratio between the available VRAM and the size of the volume will also be performed, and the results will be discussed. Finally, this chapter will also present how the library handles the rendering of a volume of 55 GB on one of the test machines.

Finally, since our implementation of the out-of-core rendering techniques showed convincing results, we started to explore the study of different surface extraction techniques. Surface extraction techniques enable the extraction of a non-voxel representation of the surface of an object from a voxel representation of it. These techniques are expected to provide a precision smaller than the size of a voxel and are useful in the metrology field to measure the features of a scanned object. This study will be presented in Chapter 5.

Chapter 6 expresses a conclusion to this work and draws a picture of the different directions to continue it.

Chapter 1

State of the art of large scale volume rendering

In this first chapter, the state of the art of out-of-core rendering, data structures used to efficiently store voxel volumes, and volumetric rendering for voxel data will be presented. The first section will focus on data structures used to store large voxel volumes in the context of both out-of-core rendering and data compression. The next section will focus on volumetric rendering in the context of CT volumes, and the last section will present future developments in hardware architecture that could be useful in the context of out-of-core rendering.

1.1 Efficient data structures for large voxel volumes

1.1.1 Sparse Voxel Octrees (SVO)

A popular data structure used to store large voxel volumes is the sparse voxel octree (SVO). A sparse voxel octree is a tree where each node contains a block of voxels and 8 pointers to its 8 child nodes. A block is a cube of voxels whose side is usually a power of 2. A leaf of the sparse voxel octree is either a bloc composed of only empty voxels, a block full of voxels of the same value, or a block that represents the highest available level of detail (LOD). A SVO is constructed from the leaves to the root, where the bloc of a parent node is typically constructed from the mean voxel values of its eight child nodes. Thus, the higher in the octree a node is, the fewer details it has. Since each block of the octree is composed of the same number of voxels and each block that is not a leaf is constructed from mean values. Figure 1.1 shows a visual representation of a sparse voxel octree.

A sparse voxel octree allows to split a voxel volume into blocs of multiple levels of details which enables the implementation of occlusion culling and level of detail selection. Indeed, since a volume inside a SVO is separated into blocs, occluding non-visible parts of the volume can be done by only fetching blocs that are currently visible by the user. To do level of detail selection, it is sufficient to fetch blocs in higher or lower levels of the tree depending on the requested level of detail.

Both techniques are useful in the context of out-of-core rendering since they both allow

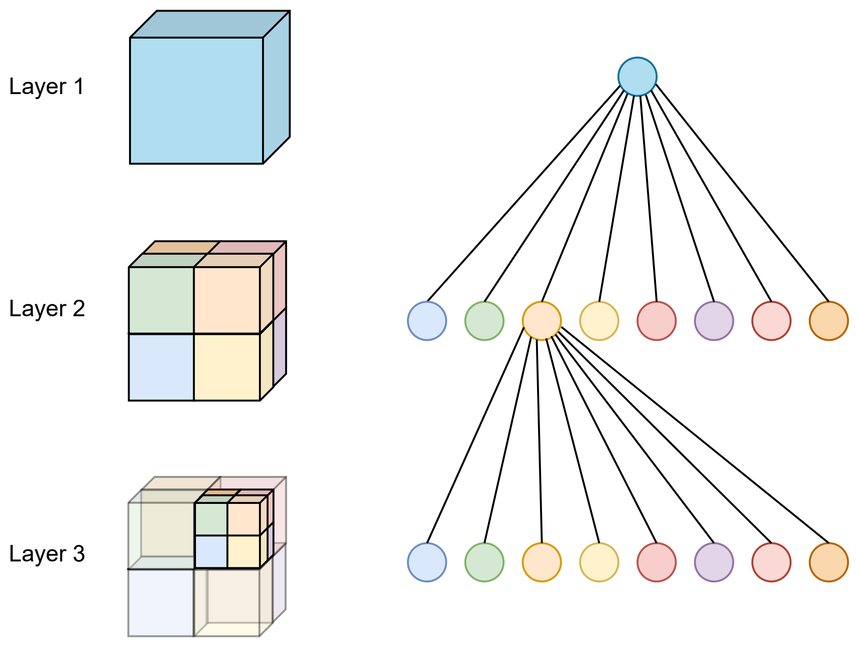


Fig. 1.1: This figure shows a visual representation of a sparse voxel octree. Each node is a bloc of voxels, and each node has 8 pointers to its 8 children. In this figure, transparent cubes represent the blocks of the upper layer. Each layer represents the same volume as the layer below it, but at a lower level of detail.

to reduce the amount of data that needs to be loaded into the VRAM for rendering. Indeed, occlusion culling enables to only transfer visible blocs into the VRAM for rendering instead of the whole volume and level of detail selection enables to use blocs that takes less memory at the cost of details for rendering. Levels of details are usually used when parts of the volume are far away from the viewpoint since the user wouldn't be able to tell the difference between the volume rendered at full details and the volume rendered with fewer details.

Crassin et al. [5] introduced a rendering and caching pipeline using N^3 trees in order to allow the real time rendering of volumes larger than the VRAM. N^3 trees are a generalization of octree where N^3 is the number of children that each node has. Octrees are N^3 trees where $N = 2$ and thus all discussed proprieties of octrees also apply to N^3 trees. Each level of the N^3 tree is the generated MIP-map of the previous level where the deepest level represents the initial full-resolution voxel model. The MIP-map algorithm was first described by Williams [7] and is a technique used to generate a series of images at different levels of details from a single image. This method is used to reduce aliasing artifacts and increase rendering performances thanks to level of detail selection at the cost of storage space. In this case, the MIP-map is generated by averaging the values of the voxels of a bloc to create the bloc of the level of detail above. The N^3 tree is the structure used to store and manage all the levels of details.

The N^3 tree structure is separated into nodes and bricks for each level of resolution. The node files represent N^3 tree structure itself whereas the brick files are used to store voxel data. Node files contain nodes which are a header for each brick of that level of resolution. They indicate if their corresponding brick is empty or not and the location of the brick data in the brick file. The brick file is just a list of voxel data for each brick.

Raycasting is used to determine which brick is visible on the screen and to determine what is the optimal level of detail that should be loaded. The optimal level of detail is considered to be the level of detail where the voxels of the brick become smaller than the size of a pixel on the screen. It's the level at which, if the level below was loaded, the user would not be able to tell the difference because of the size of the pixels on his screen. The principle behind ray casting is to cast a ray from each pixel of the screen and to determine how that pixel should be colored depending on what part of the scene was hit by the ray.

Inside the memory of the GPU, a cache pool is used to store both the bricks and the nodes that are currently used for rendering. Each time new bricks are needed for rendering, unused bricks inside the limited cache pool will be replaced based on the least recently used policy. The cache pool is managed by a page table that keeps track of the location of each brick and node inside the cache pool. The page table is used to quickly find the location of a brick or node inside the cache pool when they are needed for rendering.

Based on this paper, Crassin et al. [5] developed a toolset called GigaVoxel that allows the rendering of volumes that are larger than the VRAM of the GPU.

Engel [1] build upon the GigaVoxel paper and toolset with a focus on medical imagery and CT rendering. They added the following improvements:

– Users can load volumes of any resolution and not just ones that have a resolution of a power of two. Engel [1] idea is to use the next bigger power of 2 resolution as the resolution interpreted by the program instead of the real resolution of the model. The program will fill the new voxel data with empty voxels.

– The size of the octrees are reduced by not storing empty or homogeneous bricks. The number of child pointers used by the nodes for the bricks are reduced in order to have efficient cache traversal and a more compact tree representation.

– If more data is visible on the screen than is available in the cache pool on the GPU, the image will be refined by only rendering parts of the image that aren't at their optimal resolution. To do so a refinement mask is used to only render non-optimal bricks. This allows the tool to replace bricks in memory that are already inside the mask by new bricks that are not at their optimal resolution.

– The loading of bricks is tuned to not load bricks that can't be seen if they were hidden by the user thanks to the transfer function.

– Rendering and data loading are separated into two distinct threads which allows rendering to not be blocked by data loading.

– A color is added to bricks to inform the user if the brick has the optimal resolution or if the brick is rendered at lower resolution while waiting for the optimal resolution brick to be loaded.

– Common rendering modes were added to the tool such as direct volume rendering with transfer function, maximum intensity projection, iso-surface rendering, ...

This paper doesn't really go into implementation details since it describes the implementation of a commercially available software and thus nothing more than the general idea of the implementation can be described.

1.1.2 Directed Acyclic Graphs (DAG)

The main issue of sparse voxel octrees is their relatively large size compared to the raw voxel volume they represent because of their multi level of resolution redundancy. That is why some works are focused on searching for alternative data structures or tweaked sparse voxel octree that have less storage footprint. One of these alternatives are directed acyclic graphs.

Directed acyclic graphs are directed graphs that have fewer restrictions than trees. The main difference with trees is that nodes can have more than one parent and their children can be any node as long as no loop is present inside the structure. Indeed, in trees, a node can only have one or zero parent and there is a strict parent/child hierarchy structure. This means that if a node A is the parent of another node B, each node at the same level as B will have parents at the level of A. In DAGs, since those restrictions don't apply, there isn't a strict level hierarchy anymore as node can have any parent as long as no loop are in the structure. Figure 1.2 shows an example of a directed acyclic graph.

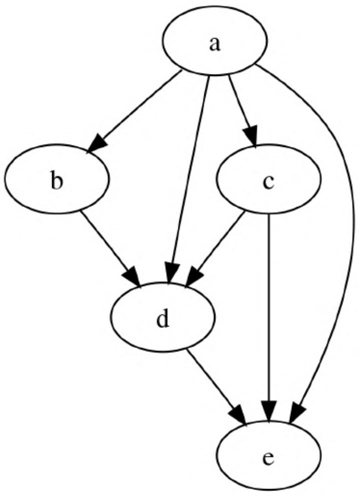


Fig. 1.2: This figure shows a basic example of a directed acyclic graph. Each node can have multiple parents and the structure doesn't have to be a strict hierarchy. But there cannot be any loops inside the structure.

Kämpe et al. [2] compare the use of directed acyclic graphs to sparse voxel octrees in the context of out-of-core volume rendering. The main benefit of directed acyclic graph is their efficiency at encoding empty regions of space, just like SVO, but also their efficiency at encoding repeating regions of space. The idea is to point bricks that contain the same voxel values to the same voxel data instead of each having their own copies of the data. This could not be done with SVO because of the strict parent/child hierarchy. A simple example of a conversion from a SVO to a DAG is shown in Figure 1.3.

This allows DAG to represent the same data as SVO while only storing unique bricks

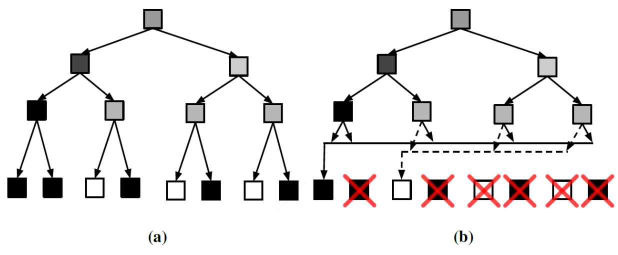


Fig. 1.3: This figure shows the steps of the conversion of a SVO to a DAG. For simplicity, the SVO is represented by a binary tree and voxels can only be black, gray or white. Figure a shows the initial SVO. Figure b shows the removal of redundant voxels in the last level of the SVO. The pointers of the voxels in the level above are modified in order for them to point to the new unique voxels. Figure c shows the same step as Figure b but this time applied to the level above. Here, the voxels pointed to must also be considered when removing redundant data. Figure d shows the final DAG. Taken from Kämpe et al. [2].

and nodes. The SVO to DAG conversion can thus be seen as a lossless compression of the SVO. In Kämpe et al. [2]'s case, they use the initial SVO as a voxel representation of the geometry of the non-voxel scene they want to render. This enables the use of rendering techniques that are voxel-based inside the non-voxel scene like rendering a volumetric smoke for example. In that context, voxels in the SVO are booleans indicating if the voxel is empty or not and the SVO uses bricks of 8 voxels.

The proposed algorithm to convert a SVO into a DAG is bottom up and the idea is to first assign a childmask to each leaf node. A childmask is a value that uniquely identify the content of a brick. Since voxels are either full or empty and that bricks are composed of 8 voxels, there are only $2^8 = 256$ possible leaf nodes and thus 256 possible childmask. The first step is to combine each leaves that have the same childmask to the same leaf node. This step is represented in Figure 1.3 b.

The next step is to go to the level above and to merge nodes that point to the same leaves nodes and that have the same childmask into the same node. The process will continue on all the levels until the root. This step is represented in Figure 1.3 c.

Kämpe et al. [2] showed that their conversion algorithm takes 45s in their worst tested case on a volume composed of 8000^3 voxels on a Intel Core i7 3930K. This is quite reasonable given the quantity of voxels and other tested volumes composed of the same number of voxels could take as low as 1s to be converted. They also showed that their DAG was able to greatly reduce the memory footprint of SVO for large voxel volumes.

They also implemented their own ray tracing and ambient occlusion computation that take advantage of their volume representation and showed improvements compared to other ray tracing and ambient occlusion implementations. But these improvements are not really relevant to the context of this thesis.

Villanueva et al. [3] go further than Kämpe et al. [2] on compression of SVO by adding symmetry awareness to DAG. The idea is to consider bricks that are symmetrically equivalent as the same bricks. This again increase the compression potential of the octree representation. Only reflective symmetries are considered and a 2D example of all possible reflective symmetries is shown in Figure 1.4. Since there is only 2^n possible symmetric transformations where n is the number of dimensions, the transformation applied to a brick can be stored with only n bits. Thus, 3 bits are stored with each brick in order to represent their orientation in 3 dimensions.

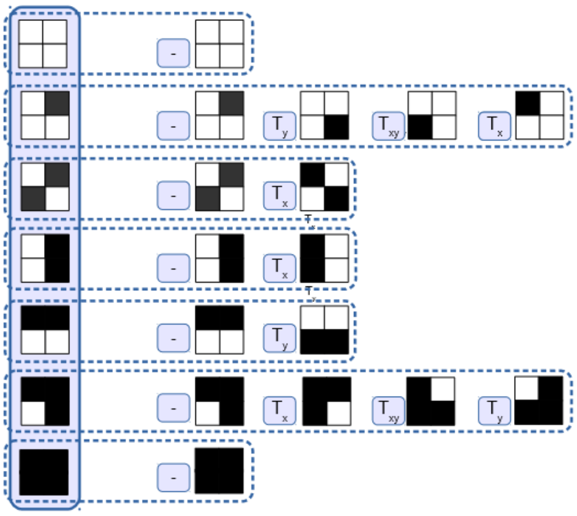


Fig. 1.4: Example of possible symmetric transformation of SADAG on a 2D square brick composed of 4 voxels. Taken from Villanueva et al. [3].

The paper shows that in some cases their SADAG had a reduction of memory consumption in the range of 66% up to 96% in the scenes that got tested. The conversion time from DAG to SADAG is also tested in the paper. The conversion time for volumes composed of 2000^3 voxels is in the range of a second and the conversion time for volumes composed of 64000^3 voxels range from 30s to 1512s. Let's note that this result does not take into account the creation of the SVO and then the conversion from the SVO to the DAG.

1.1.3 Page Table Hierarchy

Other papers have looked at a page table hierarchy to improve methods using sparse voxel octree and directed acyclic graphs to render large voxel datasets.

Page table hierarchies methods are based on the principle of virtual memory used by operating systems. The idea is to have a range of virtual memory addresses that represent the entire volume. Those virtual addresses will be mapped to physical addresses on the GPU memory thanks to a page table at runtime. A pool will be present in GPU memory where all pages will be stored and the page table will track which virtual page is currently in that cache and at which physical address that page is. When the GPU will require access to a particular page, it will use that page's virtual address inside the page table to retrieve the page's physical address. If the page isn't allocated inside the GPU cache, there is a cache miss. In that case, the page need to be send from either the RAM or the long term storage to the VRAM.

Hadwiger et al. [4] use a page table hierarchy to manage the multi-resolution voxel data that is reconstructed by a continuous stream of microscopic image data. The main appeal of virtual memory here is that there isn't a need to generate a precomputed 3D hierarchy structure and instead the necessary 3D blocks of data are only constructed from the microscopic 2D images when accessed by the GPU during ray-casting. This allows the tool to not have to deal with a long preprocessing step in order to generate a 3D hierarchy structure and also enable to render the volume while the 2D input data is still being processed to reconstruct the 3D data.

Compared to SVO techniques, there is no need for tree traversal to access data in lower tree resolutions since each level of resolution has its own page table. This reduces the latency for brick retrieval especially for bricks at lower resolution. The down-sampling ratio between each level of resolution can be chosen and isn't restricted by the octree structure.

When a brick is requested during the ray casting, only two variables l and p are required where l is the level of resolution requested and p is the 3D position of the requested brick in the range [0;1]. The required level of resolution is determined from the number of pixels taken by the brick on the user screen just like Crassin et al. [5]. With l and p, the GPU can look in his page table to get the physical address of the requested brick. If the brick is not in physical memory, a cache miss will occur.

Here there are two possibilities when the GPU has a cache miss : either the brick has already been generated or it still needed to be generated from the 2D data. In the first case, the brick will be loaded into the GPU just like any typical page table system. In the second case, the CPU will begin the construction of the brick on a separate thread and the GPU will consider the brick empty and continue the rendering loop. When the generation of the brick will be done, the new brick will be loaded into the physical memory and the GPU will be able to render the brick.

Hadwiger et al. [4] showed that their system is more scalable than the methods based on octree traversal. But one major drawback compared to the octree methods is that there is a need to wait for all the data to be computed and then transferred to the GPU in order to have a fully correct image since there is not any precomputation LOD system. In the octree methods, the needed data can just be fetched in the octree and thus the latency to render a full correct image is way lower.

Sarton et al. [6] gets its inspiration from the page table hierarchy structure used by Hadwiger et al. [4] but in the context of volume rendering instead of the rendering and generation of voxel data based on a microscope data stream. The cache mechanism used is based on the one used by Crassin et al. [5] in order to move the cache management to the GPU instead of the CPU.

One of the big objective of Sarton et al. [6] is to reduce the communication between the CPU and GPU as much as possible since it generally is one of the slowest step in out-of-core methods. Thus, in their method, the page table entities are all located in the GPU instead of having a cache dedicated to the 'node bricks' in usual octree methods. All bricks are also compressed with the LZ4 (LZ4 GitHub repository) compression algorithm. This reduces the size used by the bricks on the disk and will thus reduce the time needed to load data from the disk to the VRAM since the decompression is done on the GPU.

Their results show that their system can manage the rendering of volumes in the terabyte scale at interactive FPS and with a cache miss recovery time in the range of the tens of milliseconds in the worst cases.

1.2 Volumetric Rendering

One objective of CT rendering programs is to analyze the rendered volume to find defects or make measurements. To do so, volumetric rendering filters, rendering modes and visualization techniques need to be considered and implemented.

In this section the state of the art in volumetric rendering will be presented. A particular attention will be placed on techniques that allow the user to better see and analyse CT volumes in the context of metrology and non destructive testing.

1.2.1 Transfer functions

One of the very basic method to customize the look of a volumetric rendering is to use a transfer function. The principle behind a transfer function is to link one or multiple scalar value to a color and/or opacity thanks to one or multiple functions. These function can have one or multiple dimensions where the dimension is the number of input that the function takes. The inputs are most of the time the voxel value but using the gradient can lead to interesting results.

Chaudhary et al. [8] show that using a transfer function that links the opacity to the gradient magnitude of the voxels can be used to have a rendering that highlights the surfaces of the volume. An example of a 1.5.

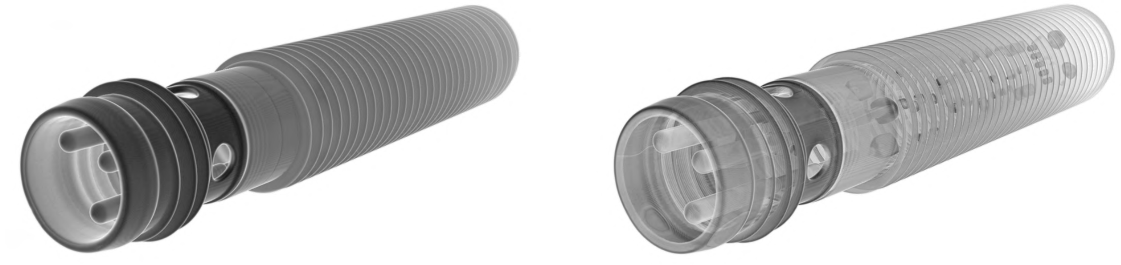


Fig. 1.5: Example of rendering with and without a gradient opacity transfer function. This image shows that using a gradient opacity transfer function can highlight the surfaces of the volume. The rendered piece is a CT scan of a proximity sensor. The colors of this image were inverted in order to remove the original black background.

The detection of surfaces from the filtering based on gradient is logical. Indeed, a surface is the separation between two different mediums. The gradient inside these two mediums is low since, inside a medium, most voxels should have values close to each others. Thus, the surface between two distinct mediums has a high gradient since each medium have voxel values in a different range.

Another way to use scalar values of transfer function is to use multi-dimensional transfer functions. The principle here is to link an opacity and color to a combination of a multitude of scalar values and can allow rendering results not achievable with 1D transfer functions.

For 2D transfer functions, Kniss et al. [9] presents the advantages of multi-dimensional transfer functions with lots of examples and also presents an intuitive and convenient interface to allow the user to define multi-dimensional transfer functions.

In this example shown in Figure 1.6, Kniss et al. [9] showed that it's not possible to isolate the bottom part of the teeth by only using a 1D transfer function on the scalar value of voxels because some voxels on the top part of the teeth have values equal to the one representing the bottom part. They then show that by using a 2D transfer function with the gradient and the scalar value as input, it's possible to fully isolate the bottom part of the teeth by not rendering voxel whose gradient isn't in the range of the top part of the teeth.

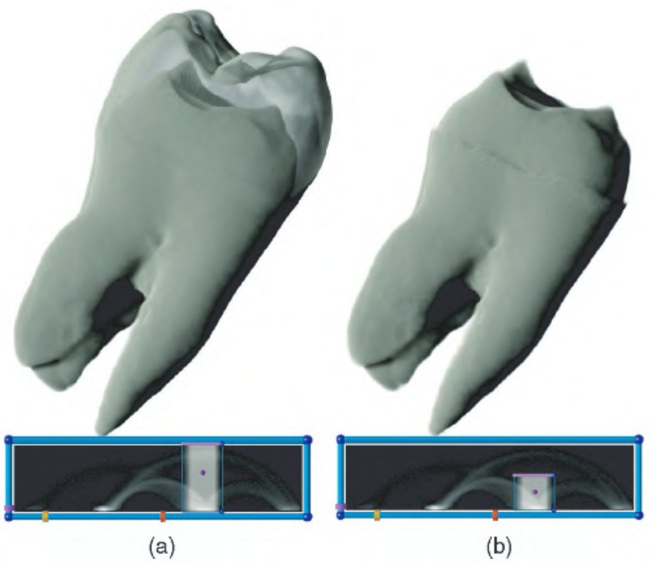


Fig. 1.6: Example of a render with on the left a 1D transfer function using scalar value and on the right a 2D transfer function using scalar value and first derivative. Taken from Kniss et al. [9]

They also show that the second derivative can be more precise and useful than the first derivative in some cases. For example in Figure 1.7, they show that by adding the second derivative as a third dimension, they can remove the imprecisions that are visible when only using a 2D transfer function with the scalar value and the first derivative of the voxel values.

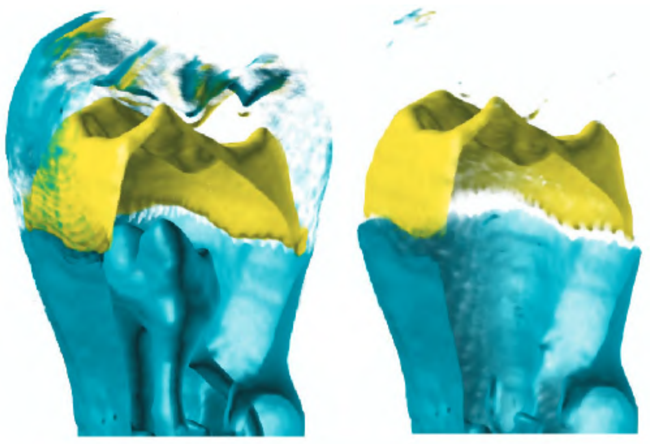
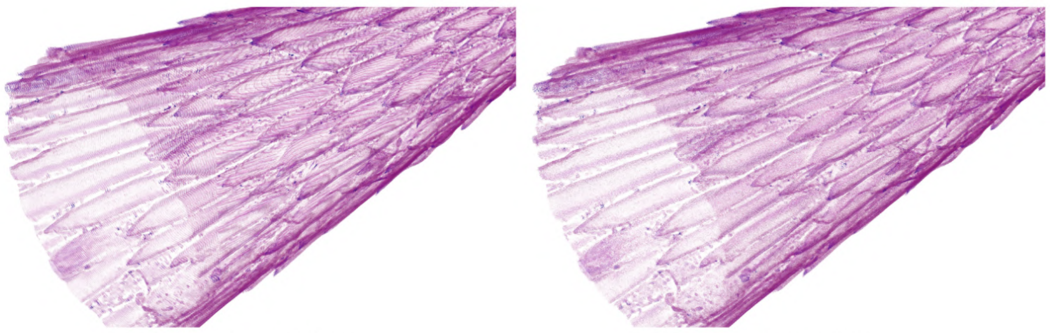


Fig. 1.7: Example of a render with on the left a 2D transfer function using scalar value and first derivative and on the right a 3D transfer function adding the second derivative. Taken from Kniss et al. [9]

The benefits of multi-dimension transfer functions on multivariate data were also presented but since multivariate data is not something of interest for this thesis, we will not discuss their findings here.

1.2.2 Rendering artifacts

Chaudhary et al. [8] show a method to reduce the aliasing artifacts without increasing the number of rays used in ray casting. The cause of those artifacts is that ray casting is sampling data from a discrete signal which is the voxel data in our case. Depending on the distance between adjacent rays, aliasing artifacts can appear on the rendering if the sampling rates are too low or identically if the rays are too far from each other. The proposed solution is called stochastic jittering and the principle is to add a random offset to the rays which will break the coherence between the neighboring fragments and thus reduces the artifacts. Figure 1.8 shows a comparison of the render of a volume with and without stochastic jittering.



(a) *Wood-grain artifacts.* **(b)** *Jittering enabled.*

Fig. 1.8: Example of a render without stochastic jittering on the left and with stochastic jittering on the right. Taken from Chaudhary et al. [8]

1.3 GPU memory architecture

In this section, technologies that are interesting in the context of out-of-core rendering will be presented. The first section will focus on the unified memory technology that is beginning to be implemented in some hardware and the second section will focus on multi-GPU systems and their potential use in the context of out-of-core rendering.

1.3.1 Unified Memory

The principle behind Unified Memory is to allow the GPU and CPU to share the same memory pool. A diagram showing the difference between usual memory configuration and unified memory can be found on 1.9. This technology would be helpful for out-of-core rendering since now the issue would not be to fit the model into the VRAM but to fit our model into the shared pool of memory that could be most of the time bigger than the VRAM of current graphics cards. Unified memory also removes the need to manually transfer data from the RAM to the VRAM since both memory spaces are combined into the same pool. This is a big advantage since the transfer of data between the CPU and GPU is one of the slowest operation in out-of-core rendering.

Integrated chips like the Apple Silicon series of processors and the Jetson series of single board computers from NVIDIA are beginning to implement this type of memory where instead of having separated RAM and VRAM, there is a pool of DRAM that can be dynamically allocated to the GPU chip or the CPU chip depending on what is needed by both. As the time of writing, Apple propose a unified memory pool of up to 192GB of memory on their M2 Ultra chip which is an order of magnitude higher that most available VRAM on current GPUs.

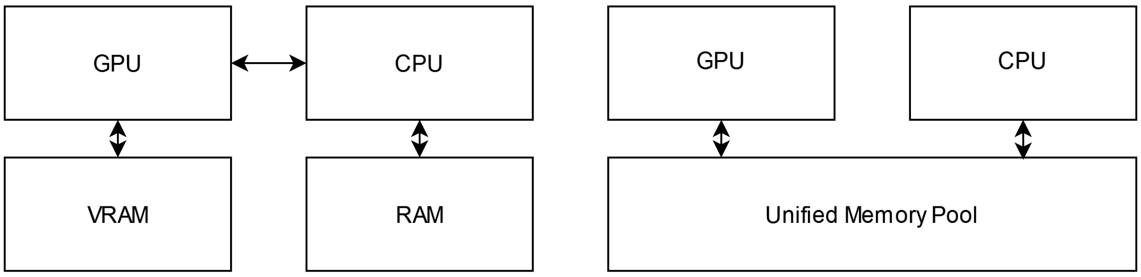


Fig. 1.9: Comparison between conventional memory configurations on the left and unified memory on the right. The unified memory configuration allows the CPU and GPU to share the same memory pool. Whereas the conventional memory configuration required data to be transferred from the RAM to the VRAM explicitly.

NVIDIA added in CUDA 6 a Unified Memory functionality that allows the CPU and GPU to allocate and manage a unified range of memory. CUDA stands for Compute Unified Device Architecture and is an API that allows developers to use NVIDIA GPUs for general purpose programming by opening access to GPU's virtual instructions and GPU's parallel computing components. Unified memory is implemented on the GPU thanks to a virtual memory system that works just like virtual memory inside an operating system. When the GPU request a page of memory that is not inside his VRAM, a page fault will occur and the page will be swapped from RAM into VRAM automatically.

From what can be found in the online documentation, this seems to be more aimed at algorithmic uses instead of rendering. It also seems that, in the context of rendering, there is a limitation in unified memory that impose that each model that is rendered must fit fully inside the VRAM. A technique like bricking would thus have to be used in order to use unified memory. The principle of bricking is to split a volume into bricks that are small enough to fit into the VRAM and to then render each block separately before combining the results of each singular render into a single full frame. Bricking was implemented by Chaudhary et al. [8] in their toolset in order to render large volumes that do not fit inside the VRAM. Though, it seems like this feature was more aimed at non-interactive rendering since the bricking process adds a lot of overhead to the rendering process.

1.3.2 Multi GPU

The idea behind a multi-GPU solution would be to use multiple GPU and either split the rendering of our model between the multiple GPUs by splitting our model in multiple parts or by unifying the VRAM of the multiple GPUs into a single large pool. The issue is that depending on the sources, it is not clear if the VRAM of multiple GPUs can be unified into a single pool with technologies like NVLINK and SLI or if the VRAM of each GPU is still separated in some way.

1.3.3 Discussion

The issue with the multi-GPU solution and unified memory in the context of out-of-core rendering is that they do not scale and only push back the issue instead of fully solving it. Multi-GPU also cannot be really considered as out-of-core since his principle is to make the memory bigger. We have shown that rendering the model in full details is mostly wasteful since all the details cannot be seen at once by the user, and we also found multiple solutions that are smarter and more efficient. Thus, we will not be looking into unified memory and the use of multi-GPU systems.

1.4 Conclusion

In this review of the literature of out-of-core rendering, a consensus was noticed. Indeed, multiple papers reviewed were based [1] or took inspiration [6, 4] from the work of Crassin et al. [5]. It was thus decided to use the GigaVoxel library as a starting point for our out-of-core rendering solution since it is open source and is distributed under the BSD 3 license which allows developers to build on top of it. In Chapter 2, the inner workings of the GigaVoxel library will be explained, and the improvements made to it in the context of this thesis will be presented.

Chapter 2

Study of the GigaVoxel library

In this chapter, our solution for the rendering of voxel volume larger than the VRAM will be presented. The solution is based on an already existing library called GigaVoxels which is based on that was already quickly introduced in the previous chapter. This chapter is divided into two main parts. The first part will introduce the library and explain its inner workings, and the second part will present the changes and improvements that were made to the library. This chapter will end with a conclusion that will summarize the work made on the library and quickly talk about how the work could be continued.

2.1 Introduction to the library

In the previous chapter, the work of Crassin et al. [5] was presented, which proposed a method using sparse voxel octrees to allow the rendering of voxel scenes that do not fit inside the VRAM of the GPU. To do so, the authors used sparse voxel octrees as a storing data structure and implemented a cache mechanism to transfer data between the GPU and CPU efficiently. Based on that paper, an open source library/tool was developed called GigaVoxels. As this library had a permissive license and seemed to fill our main requirements, it was used as a starting point for this thesis.

The library has a plugin system that allows developers to easily load a new set of functionalities on top of existing functionalities of the tool, still making use of the already available functions. The library comes with a large variety of plugins, and the most interesting one for this work is the RawDataLoader plugin. Indeed, this plugin allows converting, open and render .raw files that represent the raw binary voxel data of a model with all the useful features of the tool set. The .raw format is the format that X-RIS uses to store their 3D voxel volumes.

In the next section, the multiple algorithms and principles used in the rendering pipeline of the library will be presented and explained in details. These explanations are based on the GigaVoxel thesis by Crassin [10], the paper based on the thesis by Crassin [10] and also on the observations made while investigating the source code of the library.

Our fork of the library is available on Github[1].

[1]https://github.com/Roaanf/Roland_Greffe-Master_Thesis

2.2 Rendering pipeline of the GigaVoxel library

This section will present the rendering pipeline of the GigaVoxel library. The pipeline is divided into multiple steps that are executed in order to render a voxel volume. The pipeline is composed of 3 main steps: data storage and generation, data management and rendering. Each of these steps will be explained in detail in the following subsections.

2.2.1 Data storage and generation

In sparse voxel octrees, the voxels are grouped in cubes of voxels called bricks whose side's length is usually a power of 2 and each level of the octree represents bricks at different levels of detail (see section 1.1.1).

The splitting of the volume into bricks of multiple level of detail allows rendering only visible bricks thanks to the segmentation of the initial volume. The availability of each brick in multiple levels of details enables to render lower details bricks when they are too far away for the user to notice the loss of quality, which will save VRAM usage. The size of the bricks introduces a compromise between the overhead of the data structure and the smallest unit of data that can be managed. Indeed, the bigger the bricks, the less deep the sparse voxel octree is. But the bigger the bricks, the more data needs to be transferred to the GPU at each brick request.

The library uses its own file formats to save the octrees on the disk. There are two file types: .node files and .brick files. Each level of the octree has its own .node and .brick file. The node files are binary data that can be seen as a header file for their octree level. There, each brick will have a 32bit value that tells either that this brick is empty thanks to a well-defined value or the position of the voxel data of this non-empty brick inside the corresponding brick file. Nodes are useful since they allow us to only store non-empty bricks inside the brick files, and also allow us to place bricks in any order inside brick files. The brick file contains the raw voxel data of each non-empty brick. There are sorted by position but since empty bricks aren't stored, the node file is needed to know what portion of space is represented by each brick.

To generate the sparse voxel octree, we first need to have initial voxel data. If the volume we want to render is a voxel volume, we will directly use it as the deepest level of our octree. But if the volume is a non-voxel volume, a conversion step from triangle to voxel needs to be executed beforehand. Each upper level of the octree will then be computed thanks to the application of the MIP-map algorithm on the previous octree level. The principle of the MIP-map algorithm is to create a lower level of detail brick by taking the mean values of 8 adjacent bricks in the previous level of the octree. The resulting brick requires 1/8th of storage at the cost of voxel resolution.

Each brick also has a border of a single voxel width that is redundant data from his neighbors bricks. In future sections, when the size of a brick will be mentioned, it will always be the size without taking the border into account. For example, a brick with a brick size equal to 8 will actually have 10^3 voxels stored in it with 8^3 voxels of non-redundant data. Since the border is always a single voxel thick, the real number of voxels inside a brick will always be given by the following formula:

$$\text{real number of voxels} = (\text{brick size} + 2)^3 \tag{2.1}$$

This border is used in order to deal with boundary conditions when the hardware uses interpolation to fetch voxel values inside a brick. How and why the interpolation is used will be explained in more details in the section 2.2.3.

Once the .node file and .brick file of each level of the octree are generated, a header .xml file will be created. This file defines the size of the bricks, the name of the model, the number of levels of the octree and contains the filename of the .node and .brick files of all level of the octree. This is the file that will be used by the tool set when loading the model to know where the data is stored.

Once the generation is done, the program will initiate the rendering pipeline.

2.2.2 Data management

In order to manage the bricks and node inside the VRAM, data pools are allocated at the initialization phase inside the VRAM and will serve as a cache for nodes and bricks.

Node Pool

The node pool is used to store nodes. In the node pool, nodes are stored by groups of N^3 nodes called node tiles, where each node inside a node tile has the same parent in the octree. Here, since we are working with octrees, $N^3 = 8$. Node tiles are used in order to allow nodes to only require a single pointer to their children, instead of the 8 pointers needed in more traditional representation of octrees. Figure 2.1 represents a node pool with two node tiles inside it, where the second node tile is pointed to by one of the node in the first node tile.

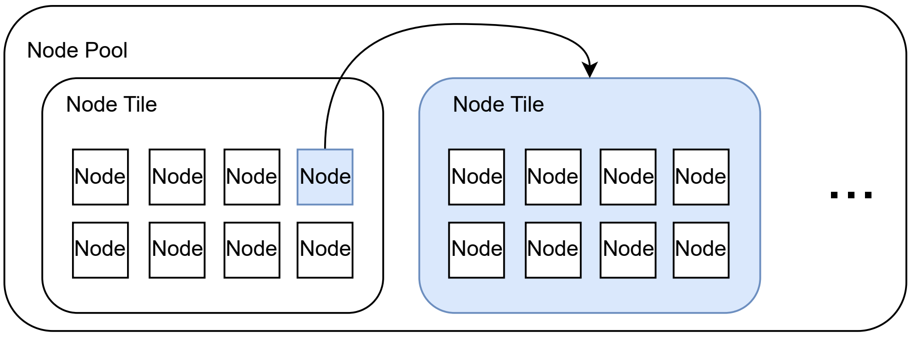


Fig. 2.1: This figure shows the structure of the node pool. The node pool is composed of node tiles, and each node tile is a group of 8 nodes that all have the same parent node in the octree. Each node contains a pointer to the node tile where all of its children are stored in the node pool.

Node inside the cache are represented by 2 32bit values, where the first 32 bit value is used to tell if the node is a leaf of the octree and the address of the node tile of its children. Only 30 bits are used to indicate the address of the child node tile, which in theory restricts the node pool size to $2^{30} * 8$ nodes. The other 32 bit value is used to either store a constant color value if the brick that the node points to is homogeneous or empty, or store the current address of the brick in the brick pool. The brick address is stored on

3 10 bits values since, as will be explained in the next paragraph, brick pools are stored in a 3-dimensional structure. These 3 10 bits values thus represent the coordinate of the brick inside that 3D structure. Figure 2.2 shows the structure of a node inside the cache.

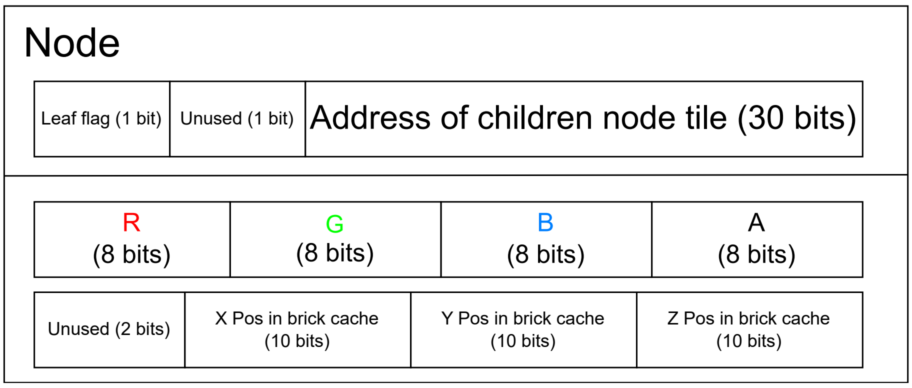

Node
Leaf flag (1 bit)
Unused (1 bit)
Address of children node tile (30 bits)
R (8 bits)
G (8 bits)
B (8 bits)
A (8 bits)
Unused (2 bits)
X Pos in brick cache (10 bits)
Y Pos in brick cache (10 bits)
Z Pos in brick cache (10 bits)

Fig. 2.2: This figure shows how nodes are stored inside the node cache. Each node is represented by 2 32 bit values, where the first value is used to store the address of the children node tile and a flag indicating if the node is a leaf or not. The second value is used to either store the position of the brick linked to that node inside the brick pool or a constant color value if all the voxels of the underlying brick are of the same value.

Brick Pool

The brick pool is stored inside a 3D texture in order to use the trilinear hardware interpolation proposed by CUDA GPUs in 3D textures. As stated in the above paragraph, nodes only use 3 10 bits value to store the location of their associated brick in the 3D brick pool. Since the pool is addressed directly through bricks and not voxels, being limited by 2^{30} bricks in the cache is not an issue. Indeed, current high tier graphics cards can have VRAM capacity of approximately 24 GB. The lowest number of voxels that can be in a brick is 8^3 voxels. If we take the border into account, this number is actually 10^3 voxels. Since each voxel stores a 2 bytes value, the maximum useable cache size would be of size $2^{30} * (8 + 2)^3 * 2 \simeq 1.95TB$ which is way bigger than any amount of VRAM that should be available in the foreseeable future. Thus, this addressing limit on the size of the brick pool is not an issue.

In this particular case, the node pool acts as a page table for the brick pool. Indeed, since node tiles represent the hierarchical octree structure and that each node in the node pool has an allocated space to store the current position of the associated brick inside the brick pool, the node pool can be used as a page table. If the node associated with the brick has a brick address in the node pool, then the associated brick can be fetched in the brick pool since its address is available. If the node doesn't have the brick address, there is a page fault that needs to be dealt with.

For the node pool, the page table is the pointers to children node tile stored in each node. Indeed, any node can be accessed from the node root by following children node tiles thanks to the octree structure. If at any point during that traversal from the root to the requested node, there is a missing child node tile pointer, a page fault occurs in the node pool. The relation between the node pool and the brick pool is represented in Figure 2.3.

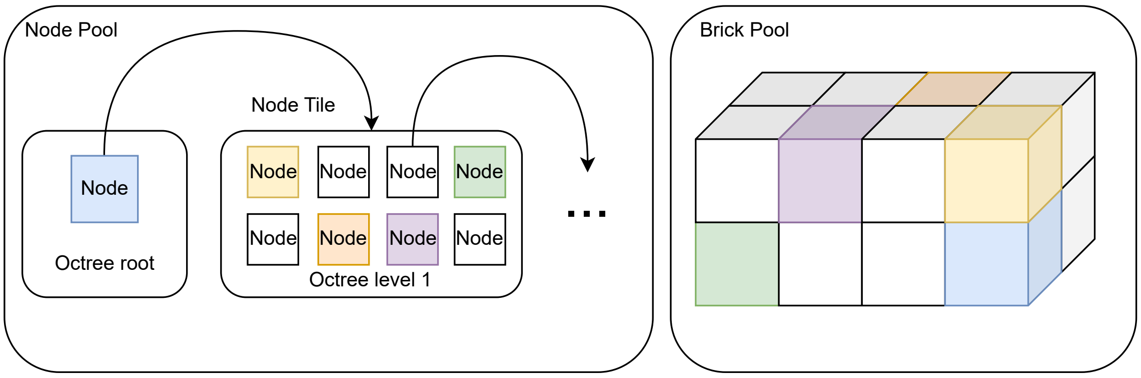


Fig. 2.3: This figure shows the relation between the node pool and the brick pool. The node pool acts as a page table for the brick pool, thanks to the pointer each node has that indicates the current position of a brick in the brick pool. This link is represented by the colors of the nodes and the bricks, where the node points to the brick with the same color. If a node is not colored, it means that its associated brick is currently not in the pool. The octree structure is implicitly represented by the parent/child relation between nodes and allows accessing any node in the node pool from the root node.

Cache management

When one of the pools is full, the least recently used replacement policy is used in order to replace unused data in the pools. To implement this replacement policy, two buffers of the same size as the page table are used: a usage buffer and a timestamp buffer. The timestamp buffer will contain for each page the last timestamp value from when the page was used. The timestamp value is a value incremented at each render pass that allows to keep track of the last time a page was used. Since the value written to the buffer is the same at each render pass, the buffer is thread safe in the sense that multiple threads using the same page will write the same value to the buffer and thus race conditions are not an issue. The usage buffer is a boolean array that is constructed from the timestamp buffer and indicates which pages were used at this render pass.

Since sorting the timestamp array at each render pass would be quite expensive, 2 order maintaining compaction are used instead. The principle of compaction is to create 2 new arrays composed only of marked elements. The first array will contain the value of the marked elements and the second array will contain the index of the marked elements. Here, the usage buffer will be used as a mask for the compaction of the timestamp buffer to get an array of the index of the elements used at this render pass. The inverse of the usage buffer will be used as a mask for a second compaction of the timestamp buffer to get an array of the index of the unused elements of this render pass. Since the compaction keep the order of the initial buffer, if the used element buffer gets concatenated with the unused elements buffer, and that the initial buffer was already sorted by timestamp, the resulting buffer will be sorted by timestamp. Since the timestamp buffer after the first compaction will be sorted by timestamp (since only two possible values will be present in the timestamp buffer), the following compaction will also be sorted by timestamp.

To deal with page faults in the caches, a data request procedure is needed. In our case, there are 2 type of request: either the children's node tile of a node is requested or the brick associated to a node is requested. Since both of these type of requests are always linked to a node present inside the node pool, the principle is to again use a buffer of

32-bit values with as many elements as nodes in the node pool called the request buffer. Here, request must be able to be emitted for each node and not for each node tile. When a thread emits a request to a node (either to retrieve the brick or to retrieve the child node tile), it will put a 32-bit value at the node position in the request buffer where the two first bits are a flag identifying the type of request and the last 30 bits represent the position of the node tile in which the request is emitted for.

Two new arrays will be created from the compaction of the request buffer, where each new list will be the compaction of all request of one of the two possible types. Again, just like the usage buffer, there is no need to check for race conditions since 2 threads requiring the same type of request on the same node will write the same value at the same place in the buffer. This is why buffer are used instead of using a vector of requests for example. One drawback of the request process is that it does not allow 2 threads to emit two request of different type on the same node. When this happens, one of the two requests will be replaced by the other and not executed at this render pass. But in the worst case, the overwritten request will be emitted again at the next render pass and is unlikely to be overwritten again since there are only two type of request and that the other type of request was served at the previous render pass.

For the Provider to manage the requests inside the request buffer, it still needs to know what data needs to be loaded in the cache. Indeed, right now, there still needs to be a link between a page table entry and a position inside the octree in order to load the correct data in the cache. To link page table entries to an octree location, 2 localization buffers are used with as many elements as there are node tiles in the page table. In our case, the first buffer encodes the position of the node in 3D space with 3 floats values between 0 and 1 and the second buffer gives the depth of the node in the tree. The localization buffers between the nodes and the bricks are shared, since a node is always linked to a brick. The localization buffers are updated at each render pass by the ray caster as will be explained in Section 2.2.3.

Now all the information needed to transfer new data into the caches is available. Indeed, the Provider knows what data it needs to fetch thanks to the requests buffer which indicates what is requested for a node and the localization buffers which provide the position of the node tile in the octree. The position of the node in the node tile is determined thanks to the index of the request in the request buffer by taking the rest of its division by 8. The location where the output of the request will be stored inside the cache is determined thanks to the LRU policy.

To serve requests, the Provider will be given both compacted request lists and fetch the requested data inside the system memory and write the n requests to the n least recently used cache positions, where n is the number of requests. In this case, both .brick and .nodes files are preloaded in system memory to speed up the data fetching process.

There is more to request serving than filling the cache with new data. Indeed, to fulfill a brick request, the Provider will have to also update the associated node in the node pool so that it points to the position of the new brick in the cache. To fulfill a children node tile request, the Provider will have to also update the parent node in the node pool in order for it to point to the new child node tile. In both case, thanks to the request buffer, the Provider will know where the node that needs to be updated is located in the node pool since the request contain the current location of the node in the node pool.

To load the .brick and .node files in the system memory, a data loader is used. The data loader will for each level search for the .node and .brick files thanks to the .xml file giving their name and location in the system. The data loader will check that the node files do have the expected size that can be determined thanks to the current level of resolution. Then data will then be loaded into memory and accessed by the Provider thanks to a given level of resolution and a relative position in 3D space.

When new data is transferred into the caches, a cache invalidation process needs to be executed. Indeed, if a cache element was overwritten during the current pass, the references still pointing to the replaced cache elements need to be invalidated. In our case, the nodes in the node pool pointing to the bricks that were replaced during the current render pass need to be changed to reflect the removal of the bricks in the brick pool.

The invalidation procedure is a two-step process. It must be done after the request have been processed because some of them might not put a new value in the cache. For example, if an empty brick was requested, no brick will be replaced in the cache and thus there is no need for any invalidation. All the requests that did replace an element in the cache will be marked as valid when they are served by the Provider.

The first step will be to take the valid requests and mark the replaced page in the page table with a replaced flag in parallel. The LRU buffer will be used to mark get the location of the replaced elements in the cache. In the second step, all elements in the page table that are pointing to the marked pages will be searched for in parallel and their pointer will be replaced by a null value to indicate that the value pointed to is no longer in the cache.

2.2.3 Rendering

In this section, the rendering process of the GigaVoxel library will be explained in detail. The first subsection will cover the ray casting process used to sample voxel values in the scene to render, and that is used to implement occlusion culling and optimal level of detail rendering. The second subsection will present the volumetric rendering techniques.

Ray casting

To determine what brick to render, a ray casting method is used. The principle behind ray casting is to launch one or multiple rays from the current viewpoint and to use the collisions of the rays with the scene to determine what to render.

Here, ray casting is also used for both brick occlusion and level of detail determination. At each render pass, a ray will be launched from each pixel of the rendering window, and each ray will keep track of all the bricks they collide with. Occlusion is thus direct, since any non hit bricks won't be in either the usage buffer or the request buffer presented in Section 2.2.2 and thus won't be rendered.

Since bricks are not necessarily fully opaque, a ray might pass through multiple bricks in a single pass. To optimize the process of ray casting, an early stop mechanism is used. Basically, a ray will, as it goes through bricks, keep track of an equivalent opacity value and an equivalent color value. Both will be updated each time the ray goes through a brick and when the equivalent opacity value gets higher than a certain threshold, the ray will be stopped early. Otherwise, the ray is stopped when it gets out of the volume.

One issue that comes from a ray casting rendering when using a single ray per pixel is called aliasing. Aliasing is a sampling rendering artifact caused by the fact that a pixel might not represent a single value in a 3D scene. Indeed, since rendering needs to output a signal (the rendered scene) on a discrete grid of pixels, the process can be seen as a sampling process. The Nyquist theorem from sampling theory thus applies, and there will be a loss of information if there is not at least 2 pixels per sampled voxel. Intuitively, the aliasing issue is caused by the fact that at a certain distance, a pixel might represent multiple voxels and if only one of these voxels is used to render the pixel, there will be a loss of information on the rendered image. An example of the possible loss of information caused by aliasing can be seen in Figure 2.4.

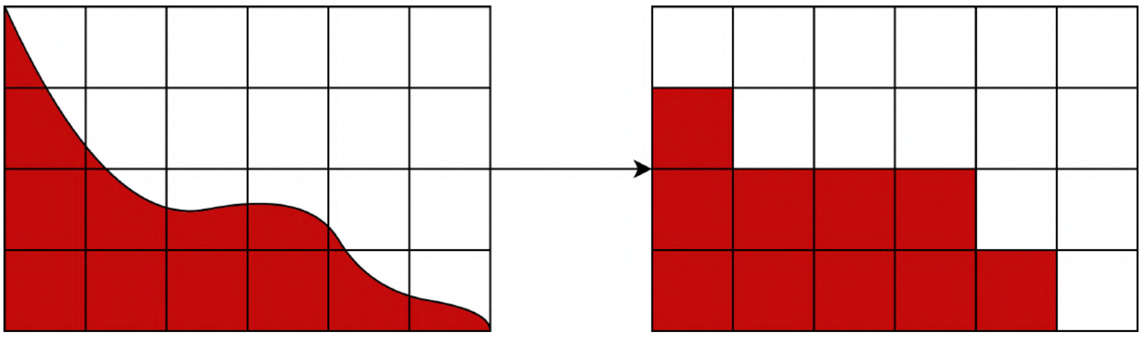


Fig. 2.4: This figure shows a simple 2D example of the possible loss of information caused by aliasing. In this case, the resulting pixel color is set to the color that uses the most space in the pixel.

Anti-aliasing is a well researched topic. Anti-aliasing techniques usually use either super sampling or multi sampling in order to reduce the aliasing artifacts. The principle of super sampling anti-aliasing (SSAA) is to render the scene at a higher resolution and to then down sample the rendered image by averaging the pixels in order to obtain a rendered image with a resolution corresponding to the screen resolution. This method gives good results but is very expensive since the higher the resolution a scene is rendered at, the higher the performance cost is required to render it. On the other hand, multi-sampling anti-aliasing (MSAA) will sample multiple values from pixel representing edges and then average all the sampled values to get the final pixel value. FXAA also uses edge detection but instead of sampling multiple values per pixel, it will apply smoothing filters on the edges in order to reduce aliasing. FXAA was first described by Lottes [11].

An example of a render with and without anti-aliasing can be seen on figure 2.5.

Here, anti-aliasing is achieved thanks to both the hardware texturing interpolation of GPUs and the optimal selection of level of detail. Indeed, since rays sample voxel data from the brick in such a way that a pixel on the screen will only sample voxels that are of similar size to it, in the worst case, a projected pixel could only represent 8 underlying voxels since otherwise the previous level of the octree that contains 8 times fewer voxels would have been selected. In that case, hardware texture interpolation is used to compute the average value of all the voxels around the sampled point. Since the resulting voxel value is averaged based on the values of the surrounding voxels, it will be an anti-aliased

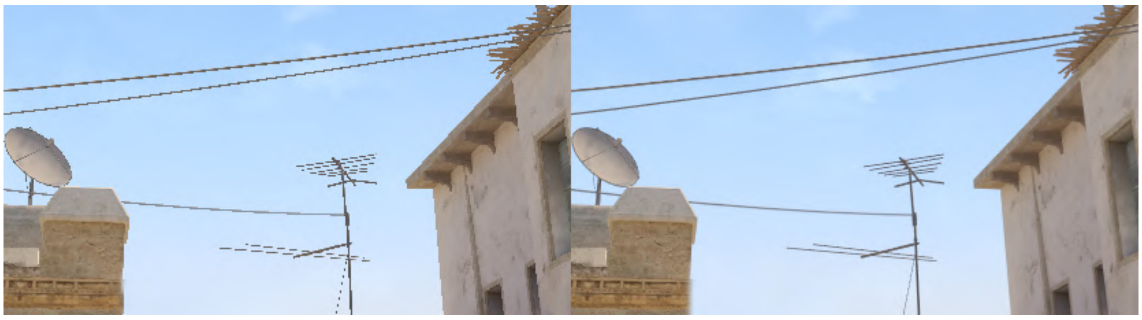


Fig. 2.5: This figure shows a comparison between a render with and without anti-aliasing. Some aliasing artifacts are visible on the left image, whereas on the right image, those aliasing artifacts are less pronounced at the cost of image sharpness. Both images were rendered at a 1920x1080 resolution, the left render did not have any anti-aliasing method enabled whereas the right render had 8x multi sampling anti-aliasing enabled. Both images are taken from the game Counter-Strike 2 at the exact same position.

representation of the volume since the sampled value will contain information from all the voxels that the pixel represents.

Since upper level bricks are MIP-mapped from the previous levels, upper levels bricks are an anti-aliased representation of lower levels bricks since MIP-maps are constructed by averaging voxel values. Thus, if a ray samples a voxel from a brick that is not at the lowest level of the octree, the sampled value will still be an anti-aliased representation of the volume.

During ray casting, each time a ray intersects with a brick, the ray will descend the octree structure in order to either fetch voxel values from the cache or to request the transfer of missing bricks into the cache. If a brick is not loaded into the cache, a request will be initiated into the request buffer and the localization information of the current node tile will be updated in order for the Provider to be able to serve that request as explained in Section 2.2.2. To descend the octree structure inside the GPU, the octree starts from the root node, which is always at the first position in the node pool. Given a node tile and the relative position of the sampled voxel in the node boundaries, it's possible to quickly determine which of the children node in the node tile contains the requested brick on the next level of the octree by only checking if the relative position is grater or smaller than 0.5 for all 3 components of the relative position. Since we start from the root node, the first relative position of the sampled voxel is its real position in the volume and the relative position is then updated at each level of the octree by applying the following formula on each coordinate component:

$$c = (c \bmod 0.5) * 2 \tag{2.2}$$

where c is the coordinate component of the relative position (x, y or z).

The descent stops either when the brick is at the lowest level of the octree, when the level of the octree is considered as optimal, or when a child node tile is not loaded into the cache. In that last case, a request will be emitted as into the request buffer and served by the Provider as explained in Section 2.2.2. The optimal level of the octree is the first level we traverse during the octree descent, containing voxels whose projected size is smaller than a pixel on the screen. This is considered to be the optimal level because at that level, if the brick was swapped by any brick with a higher level of detail, the user wouldn't be able to tell the difference because the voxels would be too small to be seen on the screen.

To test if the level of the octree is optimal, the level at which the brick is in the octree, the size of the bricks, the distance from the brick to the near plane of the camera and the field of view of the camera are needed. The distance of the brick to the near plane of the camera can be approximated by the distance that the ray has traveled from the camera to the brick. For the size that a voxel takes, the problem is simplified by always considering bricks as perfectly facing the near plane (which is the "worst" case). The level of the octree and the brick size allows determining what is the size of a voxel in the brick in the world space. The projected size of the voxel on the near place (which is the screen) is then compared to the pixel size. If the projected size of the voxel is smaller than the pixel size, The brick won't be split further and the ray will sample the voxel values directly from the brick. If the projected size of the voxel is bigger than the pixel size, the brick will be split further and the ray will continue to descend the octree structure if it can.

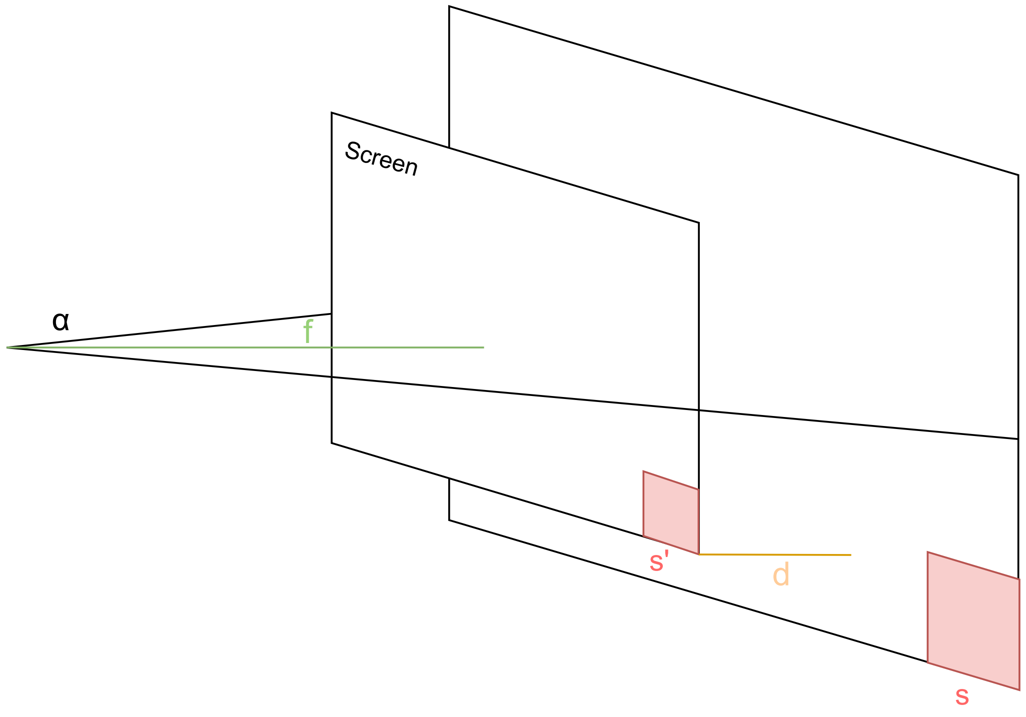


Fig. 2.6: The level of detail selection is done by comparing the size of the projected voxel in the brick to the pixel size. If the size of the projected voxel is smaller than the pixel size, the brick is considered as optimal and the ray will sample the voxel values directly from the brick. If the size of the voxel is bigger than the pixel size, the brick will be split further and the ray will continue to descend the octree structure. The size of the projected voxel (s') is determined by the size of the voxel (s), the distance of the brick to the near plane of the camera (d), the field of view of the camera (alpha) and the focal length of the camera (f).

The level of detail selection process is represented in Figure 2.6. The size of the projected voxel is given by the following equation:

$$s' = \frac{s * f}{d + f} \tag{2.3}$$

where f is the “focal length” of the camera. It’s the distance between the camera and the near plane.

Once the optimal brick is determined, the ray will march through the brick. The ray will sample the brick at a particular rate in the direction of the ray. That rate is the ray step and its value is crucial. Indeed, the smallest the ray step is, the slowest the brick sampling process and the highest the ray step is, the less sampling is done, which results in a worse rendering. The ray step value must be defined in order for it to be small enough to not skip any voxel during the ray casting. Otherwise, there will be a loss of information.

Another important feature of the ray casting is empty space skipping. The principle of empty space skipping is to skip the sampling of empty bricks of the octree and thus improve performance by reducing the number of ray samplings required. During the octree descent, if the ray finds an empty brick, the ray will skip the sampling of the brick and will directly go to the next brick. This is why the filtering of the Provider is useful to gain performance. Indeed, if the Provider filters out bricks, they will be considered as empty and the ray will skip them.

Volumetric Rendering

The library has multiple mechanisms already in place to control how the bricks will be rendered. The library allows the user to define a transfer function that maps voxel values to red, green, blue and alpha (RGBA) values. The transfer function is implemented thanks to an array containing 256 arrays of 4 float values, where the position represents the voxel value we want to map and the array of 4 float values represent the RGBA value assigned to that voxel. Here the 256 values is a limit on the definition of the transfer function and not on the represented voxel values. Indeed, the transfer function maps the normalized voxel values and not the absolute voxel values to a RGBA value. The array is present in host memory and bonded to a texture in the GPU, which will automatically use interpolation if voxel values between the 256 possible levels are requested. To apply the filtering during the rendering, a shader is used. The shader will, for each cast ray, convert the voxel intensity into a color thanks to the 1D texture of the transfer function each time the ray hits a voxel.

One other feature of the library is to allow the user to define the minimum intensity value for rendered voxels. There are two different ways to set that value: one inside a post rendering shader and one inside the Provider. The value inside the shader works by converting each intensity value of a voxel hit by a ray to the transparent value if that voxel has a lower intensity value than the one defined by the user. This method allows for very precise tuning since the shader does the filtering per voxels. The drawback is that the ray will sample voxel values that will be rendered as empty, and thus processing time is being used to render voxels that are useless to the user.

For the value in the Provider, the filtering is done before bricks are sent to the GPU. At each request for a new brick, the Provider will look at all voxels inside the requested brick and compare them to the defined value. If there is at least one voxel that is higher than the defined value inside the brick, the brick will be transferred to the cache as expected and if not, the brick will be reported as empty to the GPU pool and will not be transferred. The benefits compared to the shader method is that filtered-out bricks do not participate at all in the render pass and thus do not take resources that could be used for other non-empty bricks. One drawback compared to the shader is that if any of the voxel inside the brick is higher than the threshold, the whole brick will be rendered and sent to the GPU, which might lead to bricking on the rendered model. Bricking means that the borders of the bricks will be made visible because voxels will be rendered on one side of the border of the brick and not on the other side if the neighbor brick is filtered out by the Provider. This is because the smallest unit the Provider has control on is the bricks. To avoid bricking, the Provider and the shader should have the same threshold value. Another issue is that each time the value is changed, the cache needs to be flushed, since there might be bricks currently in the cache that should not be considering the new range.

One issue that was noticed when using the library is that with this method, the Provider only looks at the value of the current brick but does not take into account the possible voxel values that are inside the children of the brick. This is problematic because the renderer will never ask for children of an empty node. In practice, this leads to an issue where if the threshold value is too high, big parts of the volume or even the whole volume will be considered as empty because the bricks at the top of the octree do not have a value bigger than the threshold but their children do. This greatly limits the potential of the Provider threshold since the issue appears for relatively small threshold values. In Section 2.3.4, a solution to this issue will be presented.

Another parameter that can be tweaked by the user is called the "opacity distance". This is a correction term on the opacity value computed by each cast ray. The opacity value of each ray will be given by the following formula where α is the opacity distance value, β is the current opacity value accumulated by the ray and γ is the opacity value of the voxel value assigned in the transfer function. By default, γ is linear with the color value of the voxel. Meaning that the darker the voxel is, the more transparent it will be. The formula that will compute the opacity of the voxel hit by the ray is given by:

$$opacity = (1-\beta) * (1-(1-\gamma)^{\alpha * step}) \tag{2.4}$$

The first thing that can be noticed from the formula is that if either the accumulated opacity value of the ray (β) is equal to 1 or if the opacity of the voxel value in the transfer function (γ) is equal to 0, the output of the function will be equal to 0 which is expected since in both cases, we want the voxel to be considered as transparent. The other is the influence of the α parameter on the opacity value of the voxel. The α parameter is a correction term that will modify the influence of the step on the opacity value. The higher α is, the closer voxels will be considered as fully opaque. Indeed, since $(1-\gamma)$ is a value between 0 and 1, the higher the value of α is, the closer that difference will be to 0 as the voxel is further away which will lead to an opacity value close to 1. High α values will render the volume as more opaque and give a more solid look to the volume, a bit like isosurface rendering. The lower the value of α is, the less the step has an influence on the value of the opacity, and the results will be just as if γ was linear relative to the darkness of the voxel color. This will make darker voxels transparent and whiter voxels opaque. Low α values will render the volume as more transparent and give a more foggy look to the volume, since most of the time, voxel values of a volume are gray and thus semi-opaque. The final value added to the ray will be the opacity value of the voxel multiplied by the color value of the voxel for the colors and the opacity value of the voxel for the alpha value.

2.3 Improvements and changes made to the GigaVoxel library

In this section, the improvements made to the GigaVoxel library will be presented. Those improvements were made to the library in order to improve the efficiency and improve the user experience of the library. Some other improvements were bug fixes, making the library compatible with more recent versions of CUDA and C++ and compatible with unsigned short voxel values.

2.3.1 Cleanup of unnecessary code

The library was designed to be extendable and had lots of features that were not required by our use case. Some of these features used a lot of external libraries and since we wanted to compile the code using more recent versions of CUDA and C++, we decided to remove these libraries and unused features. We needed to remove the libraries because most of them were old and not maintained anymore, and thus were not compatible with the new versions of CUDA and C++.

For the library itself, the move to a newer version of CUDA was quite painless and straightforward. The only issue that was encountered was the fact that the library was using a lot of deprecated functions that were marked as such in the newer version of CUDA but still worked and that one name of a type defined by the library was used in the newer version of CUDA. The name of the type was thus changed and everything worked as expected.

2.3.2 Overflow issues

One of the most frequent encountered issues was integer overflows. Some critical variables holing big values were defined as 32-bit integers. Most of the issues were solved by changing the variable type to a 64-bit one. The most critical overflow issue was that if the brick pool was bigger than 2 GB, incorrect bricks would be fetched when rendering, resulting in artifacts as can be seen on Figure 2.7. The issue was caused by the packing of the brick coordinates.

The process of packing brick coordinates is used to store the position of a brick in the brick pool in a single 32-bit integer. The process takes the x, y and z coordinates and concatenates them in order to have a single 32-bit value. As stated in the 2.2.2 section, bricks are stored inside a 3D texture and the x, y and z coordinates each use 10 bits to be stored. The issue was that at one point, the brick packing process was done on the x, y and z coordinates of a voxel inside the brick pool and not on the coordinates of a brick. This lead to an overflow issue since the voxel coordinates are not limited to 10 bits and thus the packing process would truncate parts of the voxel coordinates in order for the value to fit inside 10 bits. This issue caused some node pointers to point to the wrong brick in the brick pool. The issue could only appear if the pool had a capacity of more than 1024^3 voxels. Indeed, if there are fewer voxels inside the pool, the packing won't lead to a loss of information since voxel coordinates would contain less than 10 bits of information. Since we are working with unsigned short voxels, the cache size that would lead to issues would be of $1024^3 * 2 = 2GB$ which corresponds to what was observed in practice.

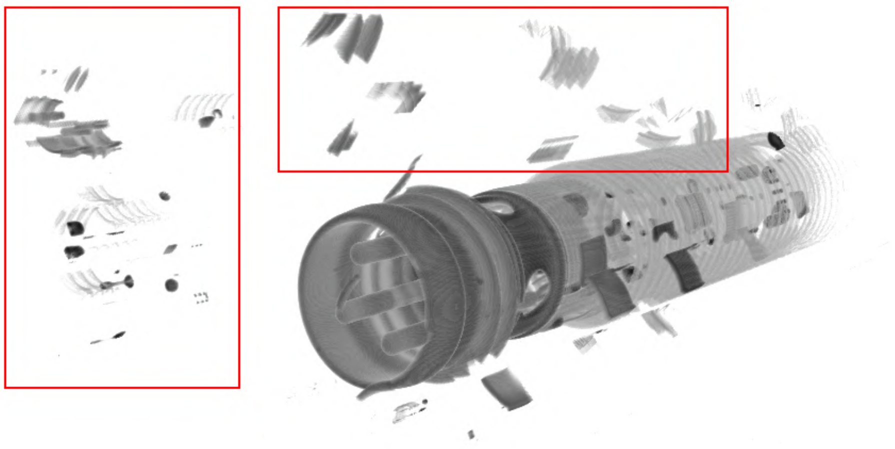


Fig. 2.7: This figure shows the artifacts caused by the overflow issue. As can be seen, incorrect bricks are fetched when rendering which leads to part of the volume being misplaced in 3D space. Some of these misplaced bricks are highlighted thanks to red rectangles. An example of the expected rendering of this volume can be found in Figure 3.2. The colors of this image were inverted in order to remove the original black background.

To solve the issue, we had to track down where the brick packing process was incorrectly used and convert the voxel coordinates into brick coordinates before packing the coordinates. This had to be done at both the packing and at all places where the incorrect packed value was unpacked, since the unpacked value was expected to be voxel coordinates and not brick coordinates. Once the issue was solved, the overflow issue was fixed and the artifacts were gone, allowing us to use brick pool sizes of more than 2 GB.

Now that the brick pool can be set to any size we want, we decided to set the size of the brick pool to 70% of the available VRAM. We do not take all of it since there still needs to be space for the node pool (but much less since node tiles take much less space than bricks) and for intermediary buffers used by the rendering process. The size of the brick pool is set at the start of the program and cannot be changed during the execution of the program.

2.3.3 Octree generation from the raw voxel volume

As already explained in Section 2.2.1, before being able to render voxel models, a conversion step needs to be executed in order to generate the octree structure of the model. In the RawDataLoader plugin, the conversion step is managed by the RawFileReader class and the conversion is separated in 3 steps: the reading of the raw file and the generation of the lowest octree level, the computation of the border of each brick and the generation of the upper levels of the octree thanks to the MIP-map algorithm. In this initial form, the generation is very slow. To give an idea, the conversion of a $840 * 1103 * 840$ voxels volume with bricks of $8*8*8$ voxels took more than 15 minutes in total, which is not acceptable for a volume of this relatively small size. In the next subsections, each step of the conversion process will be explained with an emphasis on the issues that were encountered and the changes that were made to solve them. The last subsection will act as a conclusion to the section and will give an overview of the final performance of the conversion process compared to the initial one.

Reading of the raw file and generation of the lowest octree level

Each octree level is managed through a DataStructureIOHandler object. The structure has a counter that gets incremented each time a new brick is processed, a buffer that has the size of a brick and a current brick number that indicates what is the node index of the brick currently in the buffer. A setVoxel function is available and allows setting the value of a voxel given its global position inside the model. When called, the function will convert the voxel position into a node index (since each node represents a particular region of space in the model). There are now 3 possible cases:

- The brick of the node corresponding to that voxel is currently loaded in the buffer. This can be checked thanks to the current brick number that indicates the index of the node assigned to the brick currently in the buffer. If the computed index is the same as the current brick number, the function will just set the value of the voxel in the correct position in the buffer.

- The node has not been assigned to a brick yet. This can only be the case if the node is set to empty in the node file. In that case, the brick counter will be incremented, and that brick number will be assigned to the node inside the node file. Then, the current buffer will be saved in the brick file at the position indicated by the current brick number and the buffer will be reset. The value of the current voxel can now be set in the correct position in the buffer.

- The node is already assigned to a brick. This must again be checked directly in the node file. In that case, the function saves the current buffer in the brick file at the position given by the current brick number, loads the brick in the buffer at the position given by the node file and then sets the value of the voxel in the correct position in the buffer.

The DataStructureIOHandler object is created and managed by the RawFileReader object. The RawFileReader object is the one that will read the raw file voxel by voxel and call the setVoxel function of the DataStructureIOHandler object for each voxel read. The position of the voxel in 3D space is easily computed since the resolution of the model is known by the RawFileReader. The relation between the two objects is represented in Figure 2.8.

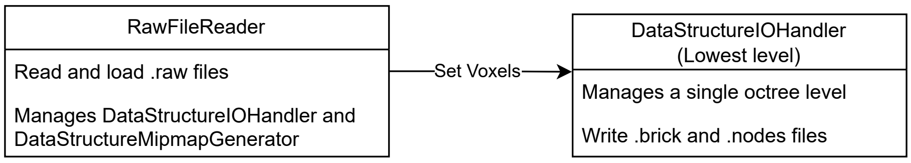

RawFileReader
Read and load .raw files
Manages DataStructureIOHandler and DataStructureMipmapGenerator
Set Voxels
DataStructureIOHandler (Lowest level)
Manages a single octree level
Write .brick and .nodes files

Fig. 2.8: This figure shows the relation between the RawFileReader class and the DataStructureIOHandler class. The RawFileReader only directly manages the DataStructureIOHandler of the lowest level of the octree. The other levels will be managed by the DataStructureMipmapGenerator object. After the reading of the raw file, the DataStructureIOHandler object will contain the lowest level of the octree and will be given to the DataStructureMipmapGenerator object to generate the upper levels of the octree and to save the octree in the node and brick files.

There are some issues with this method that lead to a long time to convert the model to an octree. The first issue is the use of a single brick buffer combined with the voxel read order of the raw file. Indeed, since the voxels are not read in the same order as bricks, the buffer needs to be swapped for each 8 voxels read if the brick has a side of 8 voxels. This led to constant IO overhead, which slows the process to a crawl. The second issue is that the RawFileReader object requires the volume's resolution to be a cube of a power of 2.

Our first change was to allow the RawFileReader to read volumes of any size. Indeed, there was a limitation to the library that only allowed it to open volumes composed of the same power of two number of voxels in all 3 directions. To remove that restriction, any inputted model will be opened, no matter the number of voxels composing it. The RawFileReader will only consider the inputted number of voxels in all 3 directions in order to correctly read the .raw file data, but the DataStructureIOHandler will be set as if the given volume was composed of the same power of two number of voxels in all 3 directions. The power of two must be selected in a way that the given volume can fit inside the "virtual" voxel cube that has a side of a power of two. Thanks to the fact that the DataStructureIOHandler object initializes the node file as only containing empty nodes and only changes a node if a non-empty brick is set, everything will work as if the model was padded with empty bricks. This will not lead to any processing time overhead since empty bricks are not stored in the brick file and that the RawFileReader only reads non-empty voxels.

The second change was to modify the order in which the RawFileReader reads the voxels to minimize the number of swaps required by the brick buffer of the DataStructureIOHandler. This is done by reading all the voxels of a brick before reading the voxels of any other brick. This allows the DataStructureIOHandler to only swap the buffer once per brick and not once per 8 voxels (if we have bricks of $8 * 8 * 8$ voxels). Since we know the size of each brick and the resolution of the model, it is easy to read the file "bricks by bricks".

While this change reduces the amount of IO overhead, the swapping of the brick buffer each time a new brick is read still represent a considerable overhead. The proposed solution was to use two large buffers representing the content of the .node file and of the .brick file. Those buffers are used instead of writing to the .node file and .brick file each time a new brick is read during the process. The size of the node file is known since it only depends on the level of resolution of the octree, but the size of the brick file is not known since it directly depends on the voxel volume itself. To solve this issue, the brick buffer is set to have the maximum size that the brick file could have, and the brick counter is used at the end of the process to only write the quantity of bricks used in the brick file. This removes the constant swapping of the buffer and thus the constant IO overhead at the cost of a far bigger memory usage.

The reading of the raw file was also changed to read the file by chunks of $x * y * brick$ voxels, where x and y are the x and y components of the size of the volume and $brick$ is the brick width. Put more simply, the file is read by slices of bricks. This was done because the initial implementation of the file reader would read the file voxel by voxel through a C++ IOStream. This method is obviously quite slow because of the overhead caused by reading a multi-gigabyte file 2 bytes at a time. To speed up this process, our first solution was to fully load the file inside a buffer in memory. Unfortunately, this leads to too much memory consumption when factoring in the buffers used by the DataStructureIOHandler. The compromise of reading the file by brick slices was found to be the best solution between memory consumption and processing time. Reading per brick slice and not voxel slices was done to easily fill the DataStructureIOHandler by bricks. Indeed, if the file was read by voxel slices, there would be a need to load multiple slices to fill up a brick.

Computation of the border of each brick

Each brick requires a supplementary border of voxels in order to ensure a correct trilinear interpolation by the hardware. Indeed, as explained in Section 2.2.2, the brick pool is implemented inside a 3D texture on the GPU. One reason for this is that 3D textures offer hardware interpolation. This means that if we request a value inside the texture between multiple voxels, the returned value will be an interpolation of the surroundings voxels. This is useful since cast rays might not hit voxels directly, but might hit between two voxels. The border is redundant voxel data taken from the brick's neighbors and is used to be able to use the interpolation at the edge of the brick.

The computation of the border is done by the DataStructureIOHandler object. In this case, 2 brick buffers are used, one that will contain the current brick and the other containing one of the 26 neighbors of the current brick. Again, in this case, the buffers use the swap system described in Section 2.3.3 that require lots of IO operations. The computation of the border will loop through all the bricks of the volume, for each brick the algorithm will loop through all the 26 neighbors of the current brick and for each neighbor it will loop through all of its voxels. For each of these voxels, a condition will check if the current voxel is on the border of the current brick. If it is, the value of the voxel in the neighbor buffer will be copied into the current brick buffer.

It is quite easy to notice that this process is doing lots of useless work. Indeed, the process loops through all the border voxels of each neighbor brick, even though only a small amount of these voxels actually compose the border of the current brick. There is no need to loop through all the voxels of the neighbor bricks since given the relative position of the neighbor brick and the current brick, we can directly know which voxels are part of the border and this is the first change that was made to the algorithm.

The first change was to remove the loop through all the voxels of the neighbors, and to instead use the relative position of the neighbor to the current brick to determine which voxels needed to be copied. This removes a lot of useless loop iterations and thus reduce the processing time of the algorithm.

The second change was to replace the use of the 2 bricks buffer by a buffer that contains all the bricks of the current level of the octree and 2 numbers that keep track of the current brick number and the current neighbor number. This removes the constant need to read and write small buffers into files on the disk which is slow and allows the object to quickly read and write brick data.

Another small change was that the computation of the border printed the position of the currently processed brick which would add some processing time that is not useful by doing lots of print command into the console. This was thus removed.

Generation of the upper levels of the octree thanks to the MIP-map algorithm

As already explained in Section 2.2.1, the MIP-map algorithm is used to generate the upper levels of the octree. The principle of the MIP-map algorithm is to create a lower level of detail brick by taking the mean values of 8 adjacent voxels in the previous level of the octree. The resulting brick will represent the average voxel values of the eight bricks that are his children in the octree structure.

The MIP-mapping is managed by a DataStructureMipmapGenerator object. This object will loop through all the levels of resolution of the octree and generate a new DataStructureIOHandler object that will manage the current level of resolution. The DataStructureMipmapGenerator object will then loop through all the voxels of the current level of resolution and compute the mean value of the 8 adjacent voxels of the previous level of resolution corresponding to the current voxel. That mean value will then be set in the current DataStructureIOHandler object at the position of the current voxel. When all the voxels are set, the previous DataStructureIOHandler object will be deleted and the border of the bricks of the current level of resolution will be computed. Afterward, the files will be written to the disk and the process will be repeated for the next level of resolution. The relation between the DataStructureMipmapGenerator object and the DataStructureIOHandler object is represented in Figure 2.9.

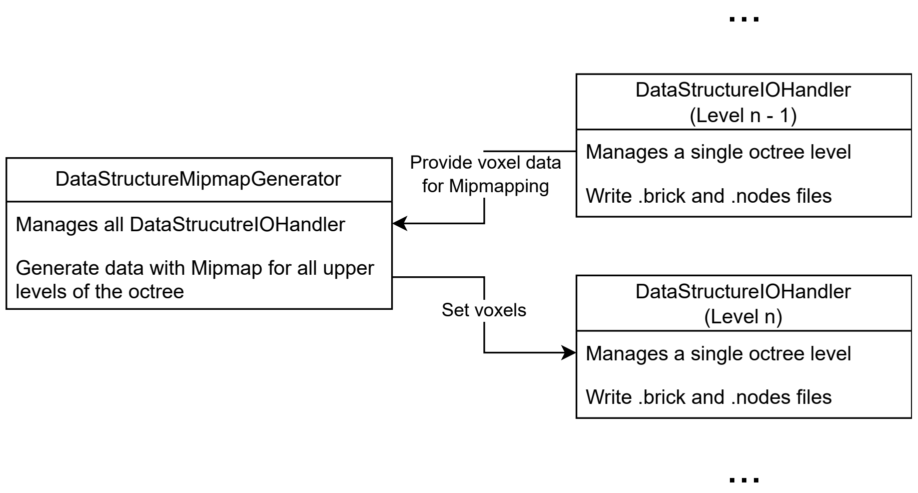

...
DataStructureIOHandler
(Level n - 1)
Manages a single octree level
Write .brick and .nodes files
Provide voxel data for Mipmapping
DataStructureMipmapGenerator
Manages all DataStrucutreIOHandler
Generate data with Mipmap for all upper levels of the octree
Set voxels
DataStructureIOHandler
(Level n)
Manages a single octree level
Write .brick and .nodes files
...

Fig. 2.9: This figure shows the relation between the DataStructureMipmapGenerator class and the DataStructureIOHandler class. Only the generation of the n level is shown, but the process is repeated for each level of the octree. When the n level is generated, the DataStructureIOHandler object of the n-1 level is deleted and the border of the bricks of the n level is computed. The files are then written to the disk and the process is repeated for all levels until the highest level of the octree is reached.

Here, the only direct change made was to remove print statements that printed the position of the currently processed brick. The changes made to the DataStructureIOHandler object presented in the previous sections also apply to the DataStructureMipmapGenerator object, since it uses the DataStructureIOHandler object to manage the octree levels. This alone was enough to reduce the processing time of the MIP-map algorithm by a considerable amount.

Conclusion and results

To test the performance gains caused by the changes described in Section 2.3.3, multiple models with varying sizes were used. The tests were made on 2 different computers with different configurations. Their specifications can be found in Appendix A. For each volume, different brick sizes were also tested for each version of the code. Indeed, the brick size has a big impact on the generation time of an octree since it directly determines the number of levels that an octree has. Because of the long time it took for the initial version of the algorithm to convert the volumes into octrees, a limit was placed on the size of the maximum volume to be tested.

Figure 2.10 shows the results of the tests. The results are the average of 3 tests for each brick size, and are given is seconds. The modified version of the conversion algorithm is on average 53 times faster than the initial version. The results also show that the conversion time of the modified version is faster when bigger bricks are used whereas on the initial version, the opposite behavior can be observed. This is because the initial version of the algorithm is more prone to IO overhead and thus the bigger the bricks are, the larger each IO operations are, but the IO operations are less frequent since there are more voxels per brick. In all cases, the bigger the bricks the less number of levels of octree might be required to represent the volume and thus the less number of levels the MIP-map algorithm needs to generate. This is why the conversion time of the modified version is faster when bigger bricks are used.

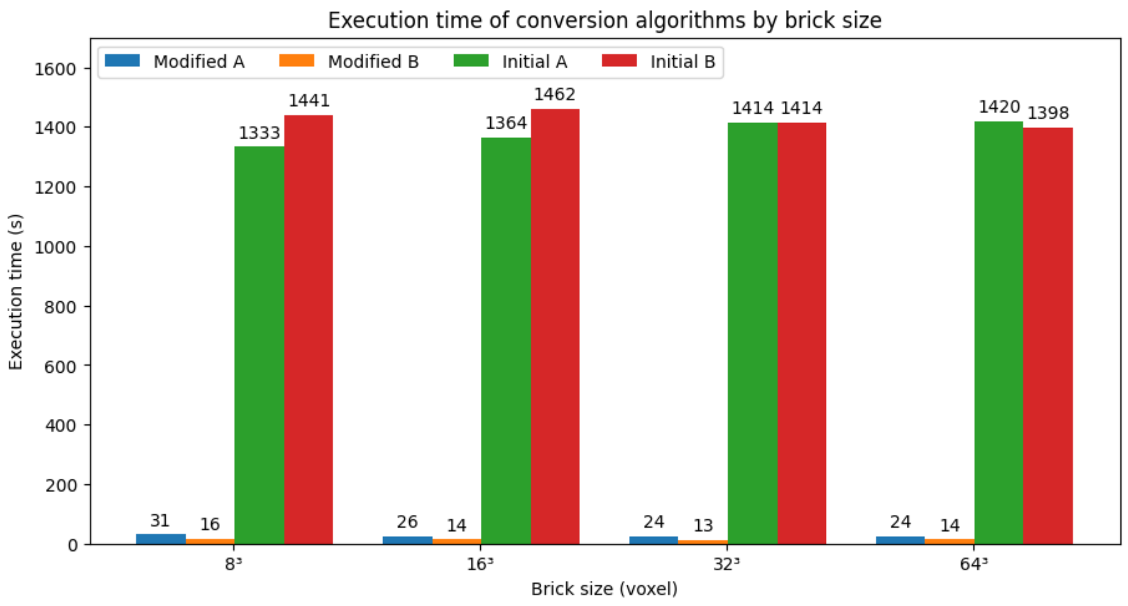
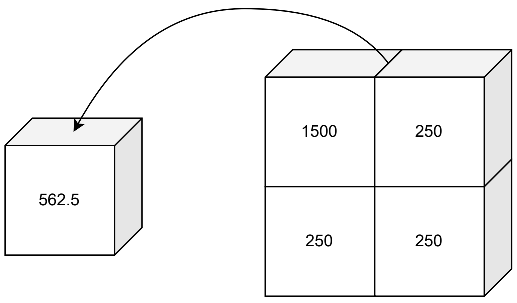


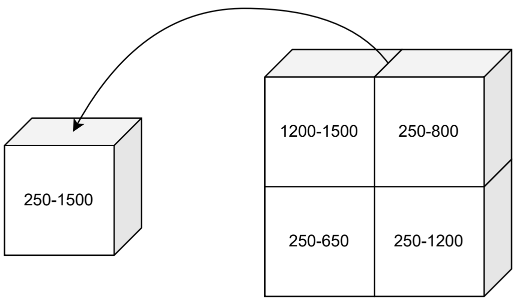
Fig. 2.10: This figure shows the results of the tests made to compare the performances of the initial version of the octree generation algorithm and our modified version. The results are the average of 3 tests for each brick size. The tests were made on two different machines, whose configuration can be found in Appendix A.

2.3.4 Improvements to the rendering pipeline

As described in Section 2.2.3, the library provides features that allow users to filter rendered data. Here, the focus will be on the Provider and shader thresholds. The first modification made to both thresholds was to replace the single threshold value by a low



(a) In this example, only voxels are considered for simplicity and the left voxel is the parent of the 4 right voxels. If the user filters in the range 1400-2500, the parent voxel will be filtered out and his child voxel will not be rendered even though its value is in the range. The parent's voxel value is the average of his children.



(b) In this example, bricks are considered since range values only make sense with bricks. Here, if the user filters out the range 1400-2500, the parent brick will not be filtered out and the top left child will be the only one not filtered among the children.

Fig. 2.11: This figure shows a comparison of the issue caused by the Provider threshold filtering and the proposed solution on a similar example.

and high threshold value. This allows the user to select any range he wants instead of limiting the filtering on only a low threshold value. The unit used for each type of threshold was also changed to be defined between 0 and 65535 which are the possible values of an unsigned short variable which is the data type used to encode the voxel values inside X-RIS raw files.

Implementation of the .range files

As already described in the section 2.2.3, one issue with the way the Provider threshold is defined is that the filtering doesn't take into account the value present in the children of the filtered bricks. In practice, this leads to an issue where bricks will be marked as empty even though they should not because a parent was also marked as empty because it was filtered out by the Provider. In the next section, our proposed solution to this issue will be presented.

The basic idea of the solution is to assign to each brick a minimum and maximum value that defines the range of voxel value both inside of the brick but also inside all of his children. This range of values would allow the Provider to check if children bricks of a node contain the value that the user wants to render, and thus solve the issue of some bricks being filtered out when they should not. An example of the issue and the proposed solution can be seen in Figure 2.11.

This is quite useful in practice because of the way volumes are captured. Indeed, a lot of voxels with small values that represent the air are present in a large radius around the scanned object. Since those voxels are not useful to the user and that there is a large amount of them, the precise filtering of these bricks will lead to a big performance gain thanks to empty space skipping.

The first step of the solution is to define a new type of file in the octree file structure called range files. Range files are binary files that will, for each level of the octree, define the minimum and maximum values of both the voxels inside the brick and in all the children of the brick for each brick in that level. This value is propagated from the deepest level of resolution to each upper level of the structure, and the files are created at the octree generation step.

At the first generated octree level, the range file generation is quite direct and easy to implement. Since the DataStructureIOHandler object loops through all the voxels of the raw file bricks by bricks, a minimum and maximum value is kept in memory and updated each time a new voxel is read. When all the voxels of a brick are read, the minimum and maximum value are written in the range file at the position of the brick. Both values are then set back to 65535 and 0 respectively, and the process continues for the next brick until all the bricks of the level are processed. This step happens at the same time as the generation of the node and brick files described in Section 2.3.3 and the added processing time is negligible.

For the next levels, the range file generation is a bit more complicated, since we need to take into account the values of the 8 children in the lower level of the octree. In the DataStructureMipmapGenerator, a minimum and maximum value is again kept in memory and updated each time a voxel is generated from the mean value of the 8 children voxels. But here the difference is that as children's bricks are traversed, those minimum and maximum values are also compared to the minimum and maximum values present in the range file of each child's bricks. The algorithm used to generate the range files is presented in algorithm 1.

The range file does not take into account the border values of the bricks. This is because since the border is redundant data from the neighbors bricks as stated in Section 2.3.3, if we filter out a brick even though a value in its border is inside the range, that value will be present in the neighbor brick that will not be filtered out. Adding the border values to the range file would thus be redundant and potentially lead to more bricks being rendered when not necessary.

After the generation of the range files, the range files will be loaded into the RAM to allow the Provider to filter out bricks based on range values. The range files are loaded in the RAM by the DataLoader just like the bricks and node files.

Now that the range files are loaded in memory by the data loader, the range information of each requested brick will also be provided to the Provider. The Provider kernel will now only check if there is an overlap between the range defined by the minimum and maximum values and the range defined by the two threshold values, instead of checking if any of the voxels of a requested brick are in the threshold range. There is thus a small gain of performance since the Provider will not loop through each voxel of the brick to check if it is in the threshold range and will instead directly compare the range of the brick to the threshold range.

The storage impact of the range files is quite low. Indeed, the range files are only composed of 2 unsigned short values for each brick of each level of the octree. This means that the size of the range files is directly proportional to the number of bricks in the octree and the number of levels of the octree. The size of the range files is thus proportional to the size of the brick file but is significantly smaller since they only use 4 bytes per brick whereas brick file require either $2 * (8 + 2)^3$, $2 * (16 + 2)^3$, $2 * (32 + 2)^3$ or $2 * (64 + 2)^3$ bytes depending on the brick size used. Indeed, in our testing, the size of the range file represented in the worst cases approximately 0.2% of the size used by the node, brick

Algorithm 1: Range determination algorithm

```
$level \leftarrow deepestLevel - 1$;
while $level \geq 0$ do
    for $brick$ $in$ $level$ do
        brick.minValue ← 65553;
        brick.maxValue ← 0;
        for $voxel$ $in$ $brick$ do
            if $voxel.value \leq brick.minValue$ then
                brick.minValue ← voxel.value
            end
            if $voxel.value \geq brick.maxValue$ then
                brick.maxValue ← voxel.value
            end
        end
        if $level \neq deepestLevel - 1$ then
            for $child$ $in$ $brick.children$ do
                if $child.minValue \leq brick.minValue$ then
                    brick.minValue ← child.minValue
                end
                if $child.maxValue$ ¿ $brick.maxValue$ then
                    brick.maxValue ← child.maxValue
                end
            end
        end
    end
    $level \leftarrow level - 1$;
end
```

and range files. In most cases, the size of the range file was even smaller than that. The storage impact of the range files and the processing time required to generate them are both negligible, but the potential performance gain is very significant.

2.3.5 Ray step

One issue noticed with the ray step is that, in its default configuration, was that it was too small and would lead to a wood grain artifact that would appear on rendered objects. This artifact is similar to the one presented in Section 1.2.2 in the work of Chaudhary et al. [8]. It is due to the fact that the ray step is too big, and thus the ray will not sample enough voxels to correctly render the object. The artifact is quite visible when the object is far away from the camera and that the volume is opaque.

In the original code, the ray step (ε) given by the following formula:

$$\varepsilon = \max(\omega * \alpha, \frac{\beta}{2^n * \lambda}) \tag{2.5}$$

ω and β are both constants in the $[0;1]$ range that are hardcoded to 0.35 and 0.5

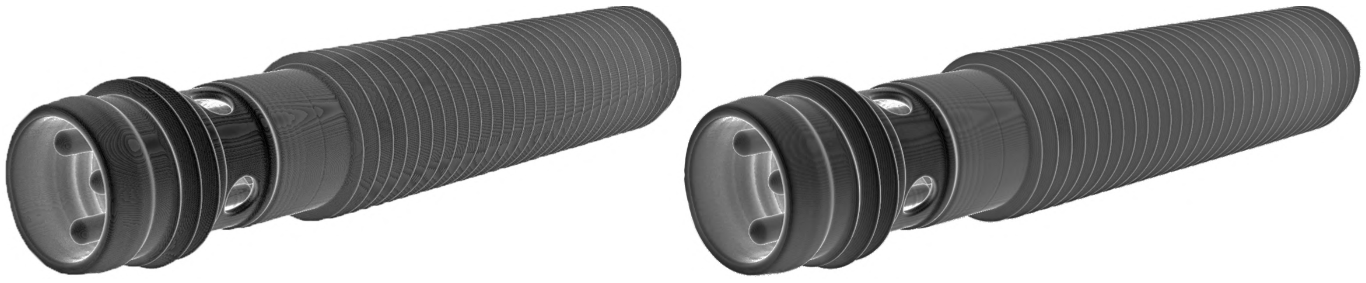


Fig. 2.12: This figure shows the wood grain artifact that appears when the ray step is too big. The artifact is visible on the left image where the ray step is too big, and not on the right image where the ray step is smaller. These artifacts appear as curves on the surface of the rendered object (which is the CT scan of a proximity sensor) and are particularly pronounced on the right part of the object. The colors of both images were inverted in order to remove the original black background.

respectively in the code, n is the level of resolution of the brick that the ray is currently passing through and λ is the resolution of the brick. The first value is proportional to the distance that the ray has traversed, and the second value is proportional to the size of a voxel inside the currently traversed brick. The first value is there to limit the ray step in cases where the ray is too far away from the camera. If the ray is too far, the increase of details thanks to smaller ray step would probably not be noticed by the user. The second value is there to limit the ray step, for it to not be too small compared to the voxels inside the brick the ray is currently going through.

Figure 2.12 shows an example of the same volume loaded and observed from approximately the same position and angle with and without our modification to the ray step.

As can be easily guessed, an increase in ray step will necessarily lead to a decrease in performance. To remedy this and because in some cases the artifacts are not visible (like when using particular rendering modes that will be presented in Chapter 3), the interface was modified to allow users to set the constant used to compute the ray step. This allows users to either lose performance in order to have less visible artifacts on the rendered image, or to gain fluidity in the rendering at the cost of the potential apparition of rendering artifacts.

2.3.6 Improvements to the toolset

Some parts of the interface were improved upon and will be presented in the next subsections.

Threshold range

The interface and the threshold were modified in order to allow the user to define both a low and high threshold value instead of only a single threshold value. Initially, only the lower value could be chosen and every voxel whose value is below the threshold would be filtered. Now, the user can define a range of values that will be rendered. Ranges also were defined in the range $[0; 1]$. This was changed to $[0; 65535]$ since the voxel values that we use should always be encoded in an unsigned short variable. This is also more intuitive for the user.

These changes were made to both the Provider and the shader threshold. The Provider threshold was also modified to take into account the range values of the bricks, as explained in Section 2.3.4.

Radius filtering of the volume

Because of the way volumes are captured, a cylindric volume is often captured at the border of the volume. Since this volume is not part of the scanned volume and is only an annoyance for the user, a radius filter was implemented in the initial generation of the octree. This filter will ask for an outer radius that will set all voxels outside the radius to 0. This is also useful to remove unwanted air voxels that are present around the volume. The radius filter considers that the volume is oriented in such a way that the axis of the cylinder that will be kept is the y-axis. This is because volumes are scanned in that way in practice.

This was done at the generation step because it is easier to do it at this step than to do it at the rendering step. It also reduces the file size of the octree on the disk, since the volume is cropped before the octree is generated. It also makes the generation slightly faster since there are fewer voxels to read from the raw file and to write to the brick file, but if the user inputs a radius that is too big, he will have to regenerate the octree himself.

Automatic scaling of the color

As stated in Section 2.2.3, the color of a voxel in a rendering is defined thanks to a transfer function. To deal with the multitude of possible voxel value ranges, the mapping uses normalized voxels values in the range $[0; 1]$. The issue is that the mapping doesn't take into account the range of value that is actually rendered. This leads to a loss of color contrast when the range of non-filtered out values is small. To remedy this, the color mapping was modified to take into account the range of values that are actually rendered. The color mapping will now take into account what range is currently rendered and will scale the color of the voxels accordingly. This is done by multiplying the normalized voxel value by the size of the range of values that are rendered, and then adding the minimum value of the range. This will ensure that the maximum color contrast is used for the rendered values. In practice, to avoid rendering fully black or white voxels, the scaling is done on the range $[0.1; 0.9]$ instead of $[0; 1]$.

2.3.7 Future improvements

Size of the octrees files

Something that was noticed during our tests was that the total amount of storage used by the octree structure compared to the raw volume size can have an increase of as much as 150% and sometimes a decrease of as much as 90%.

The decrease can be explained by the fact the in our brick files, values of 0 are not stored whereas they are in .raw files. For the increase, it is explained by the overhead of the multiple levels of resolution and the addition of .raw and .range files. While this wasn't considered as an issue, potential solutions to decrease the size of the octree structure will be presented.

The first idea is inspired by Sarton et al. [6]. The authors proposed using the lz4 compression algorithm in order to reduce the size of the octree structure on the disk and to decompress the data when sending it to the GPU. This would both reduce the size of the file on the disk but also reduce the transfer time of the brick from the CPU to the GPU since the quantity of data that needs to be transferred would be reduced. There would be a small overhead caused by the decompression, but we think that it would be greatly compensated by the reduced transfer time of the brick from RAM to VRAM.

Another idea is to implement either directed acyclic graph or symmetry aware sparse voxel octree which were both described in Section 1.1.2. These solutions would reduce the size of the octree on the disk greatly, but wouldn't directly reduce the transfer size from CPU to GPU. There would be a reduce in transfers if we introduced a system that would detect if a brick that needs to be transferred into the VRAM is already present because it's already used by another node and would thus not be transferred. This would reduce both the VRAM usage and the potentials transfers from RAM to VRAM but would require an addition to the cache system that would allow identifying duplicate bricks.

Implement the use of Nvidia's hardware accelerated ray tracing

Since the end of 2018, Nvidia has included ”RT-Cores” inside their RTX line of GPUs. These special cores are accelerator units that are designed to efficiently perform ray tracing operations. Since the method proposed here is heavily reliant on ray casting, there could be a benefit in the use of these cores in order to improve the rendering performances in large models with lots of transparency.

The issue is that RT cores can only be used through specific APIs like DirectX Raytracing, Vulkan raytracing and Nvidia's Optix library. The first two are directly integrated into a rendering API that we don't use, and Optix seems like a good choice, but we didn't take the time to investigate its implementation further. Indeed, from multiple sources online, it seems like RT cores are only useful for mesh intersection and would thus require a mesh conversion from a voxel volume into a mesh volume in order to be able to use the RT cores. The issue is that once converted, the volume would only have its surface information left, which is not very useful when doing nondestructive part analysis. Indeed, during nondestructive part analysis, the point is to be able to detect issues present inside an object.

Surface extraction is something that is going to be looked into in Chapter 5 but not in the context of out-of-core rendering. And even if it was, RT cores would only enable the acceleration of the rendering of a ray-traced scene with the surface model.

A close attention should be kept to Optix to see if one day they will allow for the usage of RT cores on voxel volume in the context of ray casting.

Brick size configuration

Right now, the brick size used by the software requires a recompilation of the whole library in order to be changed. The issue stems from the SampleCore class having the brick size as a typedef and using this type as a template class multiple times. Since the SampleCore class is the class responsible for the loading of the volume, the solution would be to either read the file outside the class or let the user define the requested brick size and then make the SampleCore class take the brick size as a template.

The issue is that changing the SampleCore class to take a template type would require a long and tedious rewrite of a lot of class definitions which we decided to not be worth the time given the relatively small necessity to change the brick size outside of testing as will be shown in Section 4.2.

A solution would be to use the plugin system that is already implemented in the library and compile the plugin multiple times with different brick sizes. This would allow the user to choose the plugin that corresponds to the desired brick size. Unfortunately, this would require 4 compilations of the plugin each time a change is made in it and would require the user to know which plugin to use. This is not a very user-friendly solution, but it would be the simplest to implement.

Reuse of previous frame information

One feature that would be quite interesting to include would be to let the renderer reuse the previous frame if the position of the camera or any rendering parameters haven't been changed by the user between the two renders and if the bricks are at their optimal level of detail. Right now, the ray casting is done at each frame, even when we know that the result will be the same. This is wasting GPU processing for nothing, and it's a feature that is already implemented in a quite a few rendering programs, like Paraview for example.

It would also be directly useful since it is not uncommon in the context of analyzing an 3D volume to stare at a particular position of a render for some time while analyzing what can be seen.

2.4 Conclusion

In this chapter, the inner workings of the GigaVoxel library and the changes made to it were presented. The changes were made to make the library compatible with the type of raw files used by X-RIS, to patch bugs, to improve the user experience and the performance of the library. Chapter 3 will present the implementation of volumetric rendering modes to the library. These modes are the ones implemented in Maestro, which is the CT rendering software of X-RIS and are designed to ease Non-Destructive Testing.

Chapter 3

Volumetric Rendering for tomography data

In this chapter, the port of the Maestro CT software rendering modes into the GigaVoxel library described in Chapter 2 will be presented. These rendering modes are specific to the field of metrology and Non-Destructive Testing and are designed to help users with their analysis of CT data. One of the goals of this chapter is to show that the bricking and caching system does not restrict the implementation of those rendering modes. The performance impact of each rendering mode will also be analyzed and compared to the default rendering mode of the library.

3.1 Gradient rendering

The concept of gradient rendering is to render the gradient of the attenuation values of the voxels instead of the attenuation values themselves. This is useful to amplify all the surfaces where a change of material is present inside the observed volume.

Two options were available for the implementation, either add a step during the octree generation that would compute the gradient of each brick and store them in a gradient file separately from the bricks and node files, or compute the gradient on the fly during the rendering. The second option was chosen since the first one would have added a considerable amount of processing time to the octree generation and, at worst, doubled the size of the brick files. One worry for the second option was the added processing time of the gradient computation during the rendering. But, as will be talked about in Section 3.3.1, the added processing time, while not negligible, is reasonable.

3.1.1 Computation of the gradient

The gradient is computed in the Provider kernel when a requested brick is loaded from the cache. This allows gradient rendering to be used with the cache system and shaders effortlessly since, as far as those objects are concerned, they are receiving the bricks of the octree. To compute the gradient, the Provider kernel will loop through all the voxels of the brick and compute the gradient with a central difference scheme for all voxels that aren't on the border. At the borders, a forward difference is used instead of a central difference.

Since the gradient is computed per bricks, there might be an error made on their border as the information from the neighboring bricks are not available during the computation of the gradient. But since bricks have voxel data from their neighbors as a border, the gradient value of the voxels inside the brick are correct.

Since we are in 3D, the gradient is a vector of 3 components. To convert the gradient to a scalar value, the magnitude of the gradient vector is computed and used as the voxel intensity value. To compute the gradient, the following formula is used for voxels not at the border of the brick:

$$\nabla f = \begin{pmatrix} |f(x+1,y,z) - f(x-1,y,z)| \\ |f(x,y+1,z) - f(x,y-1,z)| \\ |f(x,y,z+1) - f(x,y,z-1)| \end{pmatrix} \tag{3.1}$$

For those at the border, the following formula is used:

$$\nabla f = \begin{pmatrix} |f(x+1,y,z) - f(x,y,z)| \\ |f(x,y+1,z) - f(x,y,z)| \\ |f(x,y,z+1) - f(x,y,z)| \end{pmatrix} \tag{3.2}$$

After the gradient is computed, the magnitude of the gradient is computed with the following formula:

$$|\nabla f| = \sqrt{(\nabla f_x)^2 + (\nabla f_y)^2 + (\nabla f_z)^2} \tag{3.3}$$

In practice, since we are working with unsigned short values, there is a need to convert the gradient magnitude to a whole number. The ceiling of the magnitude of the gradient is thus used as the intensity value of the voxel.

Something to consider while using gradient rendering is the range values presented in Section 2.3.4. Indeed, the range values attached to each brick are the range of values inside the brick before the computation of the gradient. Thus, the range values are not accurate when gradient rendering is enabled.

There are two options to solve this issue: either revert the Provider so that it doesn't take the range value into account when using gradient rendering, or let the Provider continue to work on the defined range values. The first option will filter out bricks whose gradient values are not in the range of the user's interest, but it will suffer from the issues that were solved by the range files. These issues, presented in Section 2.3.4, are not negligible since they restrict the efficiency of the Provider filtering system.

For the second option, we could either try to recompute the range value or keep using the current range values when gradient rendering is enabled. The range values cannot be easily changed at runtime, since range values need to take into account all the children of

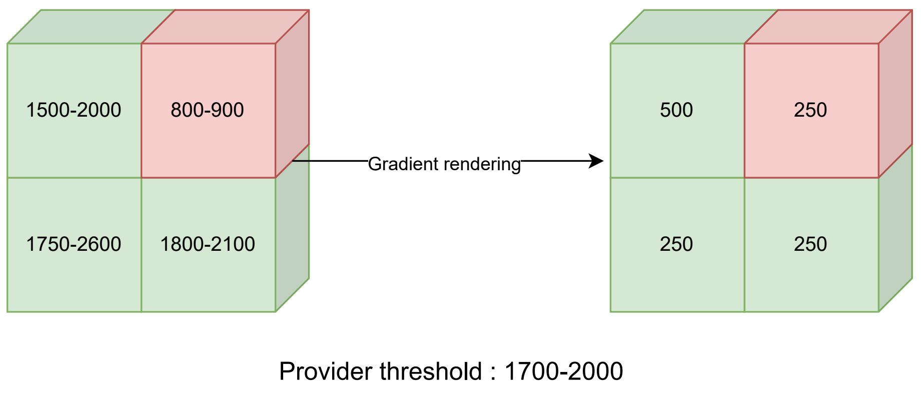


Fig. 3.1: This figure shows how the Provider threshold works when gradient rendering is enabled. The left image shows the range values of the bricks and the right image shows the gradient values of the bricks. The green bricks are the bricks that are visible to the user, and the red bricks are the bricks that are filtered out by the Provider threshold. As can be seen, the Provider only considers the range values when filtering out bricks.

a brick. But since gradient bricks are only computed when needed, the gradient children of a brick are not necessarily available when computing the gradient of a brick. If the gradient was computed during the octree generation, the range values could have been computed at the same time. But, as stated before, this would have added a considerable amount of processing time to the octree generation, which we want to avoid.

It was decided to keep the Provider threshold working on the values of the non-modified bricks when gradient rendering is enabled. This was done because the possible performance gains of the Provider threshold in highly transparent scenes are too important to not have. Especially when considering the performance loss caused by gradient rendering. Unfortunately, this leads to a quite confusing experience for the user since the Provider threshold values are applied on the range values which are the voxel values before the computation of the gradient values. In practice, this means that the gradient rendering will only be applied on the bricks that are visible to the user when he enables gradient rendering. This is not necessary an issue, but something that the user must be aware of. Figure 3.1 illustrates how the Provider threshold works during gradient rendering.

For the shader threshold, since the brick themselves get converted before being sent to the GPU, the threshold works on the gradient values. This means that the user can still filter out the gradient values that are not in the range of his interest, but the user must be careful when enabling gradient rendering since the gradient values could be filtered out by the value currently set in the shader threshold. This is not a big issue since the user can easily change the shader threshold value to see the gradient values that are filtered out, but again, this is something he must be aware of.

The different behaviors of the two threshold is where the confusion can appear. Indeed, as stated in Section 2.2.3, to avoid bricking, the shader threshold should be set to the same value as the Provider threshold. But since the Provider threshold works on the range values and the shader threshold works on the gradient values when gradient rendering is enabled, setting both thresholds to the same range might lead to some brick that would be of interest to the user to be filtered out because of the currently set shader threshold.

3.2 Rendering Filters

In this section, the multiple added rendering modes will be presented. All of these modes are the ones available in the Maestro CT software and were implemented to show that the cache and bricking systems do not restrict the implementable rendering modes. Also, all the rendering modes are implemented in the shader applied during ray-casting which allows gradient rendering to be used with all of them without any changes.

The shader is the code that deals with converting values read during ray-casting into the colors that will be rendered on the screen. Inside the shader, each ray has a color and opacity value that represent the current accumulation of information it has currently received. Each time a ray sample the voxel scene, the shader will fetch the attenuation value of the hit voxel. If that value is not in the range defined by the shader threshold, the ray skips the current voxel and travels to the next sampling point. If the value is in the range, the shader computation continues and, depending on the rendering mode, the color and opacity of the current voxel will be computed and added to the ray's color and opacity.

3.2.1 Default rendering

The default rendering mode is the simplest rendering mode. It is the mode that was already implemented in the library before any of the other ones. The idea is to convert the attenuation value of the voxel to a color thanks to the transfer function while applying the opacity correction explained in Section 2.2.3. The color and opacity of the voxel are then added to the ray's accumulated color and opacity. Here, early ray stopping is enabled. This means that if the cumulative opacity value of the voxel is grater than a certain threshold close to 1, the ray will stop and the current color and opacity of the ray will be used on the rendered pixel. This is done to save processing time, as voxels that are behind opaque voxels are not visible to the user.

Figure 3.2 shows an example of a volume render using the default rendering mode with and without gradient rendering enabled.

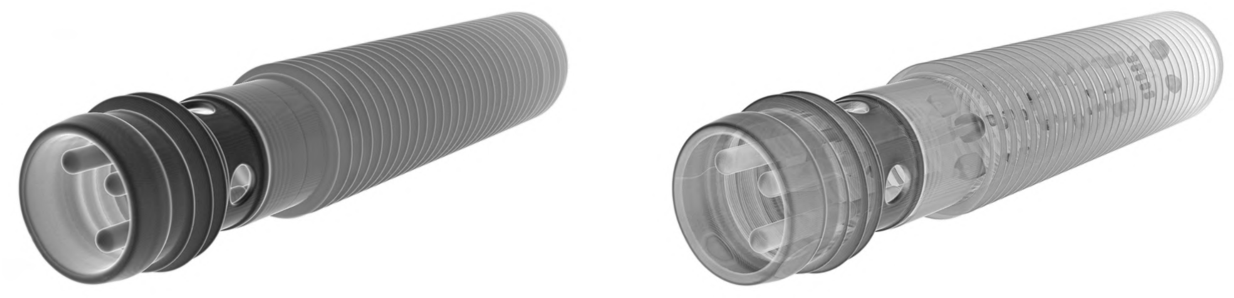


Fig. 3.2: This figure shows the rendering of a volume with the default rendering mode. The volume is a CT scan of a proximity sensor. The left image is the rendering without gradient rendering enabled, and the right image is the rendering with gradient rendering enabled. The colors of this image were inverted in order to remove the original black background.

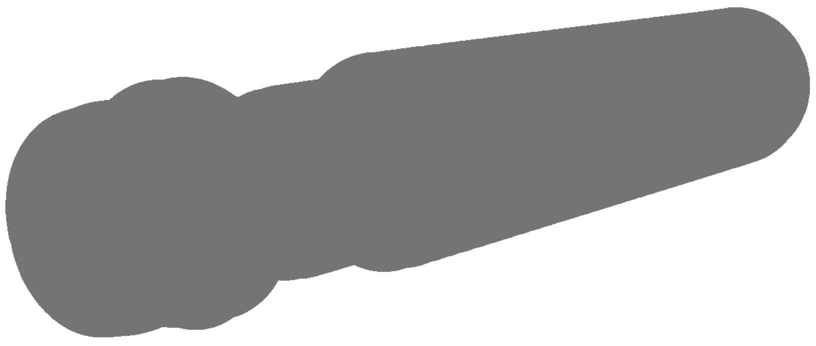


Fig. 3.3: This figure shows the rendering of a volume with isosurface rendering where all voxels have the same color and opacity values. The volume is a CT scan of a proximity sensor. As can be seen, the details of the volume are not visible.

3.2.2 Isosurface rendering

The principle behind isosurface rendering is to only consider the first voxel that the ray intersects with and to render it without directly taking its attenuation value into account. This is used to easily observe the surface of the object that the user want to observe.

The idea is to just stop the ray casting at the first intersection between the ray and a voxel. The color and the opacity that will be applied to the voxel is predetermined and will be applied to the ray's color and opacity values.

A post-processing step is still required because, right now, all the voxels will have the exact same color. This leads to the details of the volume to not be visible, as can be seen in Figure 3.3. The volume used in the figure is the one visible in Figure 3.2.

To solve this issue, diffuse lighting is used. The idea of diffuse lightning is to compute the angle between the normal vector at the voxel of the surface and the direction of the light source. In this case, since no light source are present, we consider the position of the camera as the location of the light source. The lower the cosine of the angle between the two vectors is, the darker the color because the more angled a surface is to its light source, the less light will be reflected back to it.

The normal vector of the voxel is computed thanks to a central difference scheme, just like how the gradient of a voxel is computed. Then, the angle between the normal vector and the direction of the ray is computed, and its cosine is used to compute the color of the voxel. The process is then directly stopped, and the color is applied to the ray's pixel. Figure 3.4 shows an example of a volume render with isosurface rendering.

3.2.3 Max attenuation rendering

The principle behind max attenuation rendering is to use the maximum value that the ray intersects with as the sampled value. To be able to do this type of rendering, early ray stopping was removed in order for the ray to traverse the whole volume. The attenuation value per ray is changed each time the ray samples a voxel if that voxel value is greater

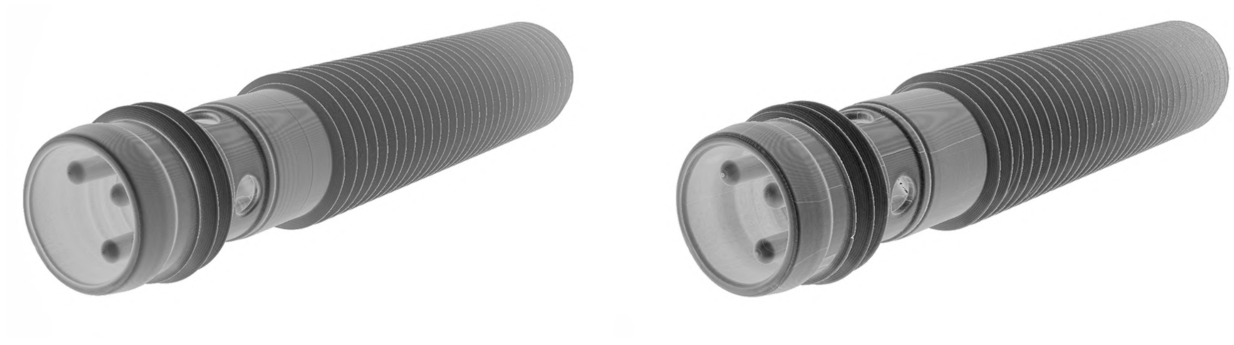


Fig. 3.4: This figure shows the rendering of a volume with the isosurface rendering mode. The volume is the CT scan of a proximity sensor. The left image is the rendering without gradient rendering enabled, and the right image is the rendering with gradient rendering enabled. As can be seen, gradient rendering is not very useful in this rendering mode, since the result in both cases is practically the same. The colors of both images were inverted in order to remove the original black background.

than the current stored value.

Because of the removal of early ray stopping, there is a small performance hit when using this rendering mode, since the ray will always traverse the whole volume.

Figure 3.5 shows an example of a volume rendering with max attenuation rendering.

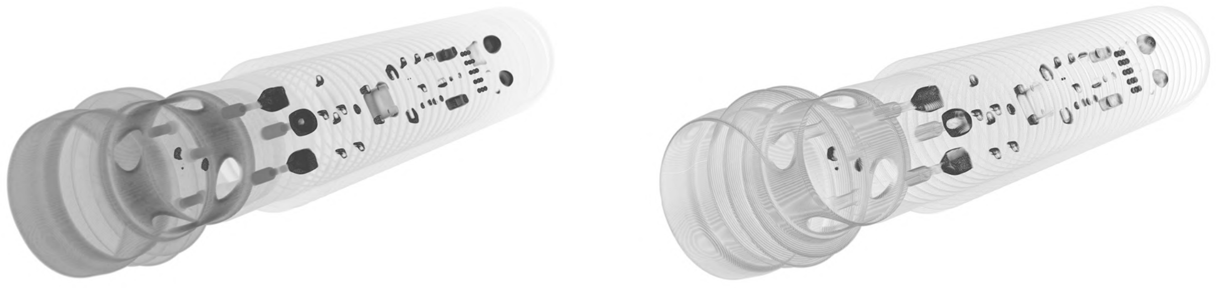


Fig. 3.5: This figure shows the rendering of a volume with max attenuation rendering. The volume is a CT scan of a proximity sensor. The left image is the rendering without gradient rendering enabled, and the right image is the rendering with gradient rendering enabled. Here, gradient rendering is not very useful, since the results with it enabled closely resemble the result without it enabled. The colors of both images were inverted in order to remove the original black background.

3.2.4 Mean attenuation rendering

The principle behind mean attenuation rendering is to use the mean value of each voxel value that the ray intersects with as the sampled value. To do so, each ray will keep a counter of the number of samples it has done, and the sum of all hit voxel values. Each time the ray intersects with a new voxel, the voxel value is added to the sum and the intersection counter is incremented. The ray will then act as if it hit a voxel with a value equal to the sum divided by the intersection counter.

Here, early ray stopping is also disabled and there is thus again a small performance loss compared to the default rendering mode.

Figure 3.6 shows an example of a volume rendering with mean attenuation rendering with and without gradient rendering enabled.

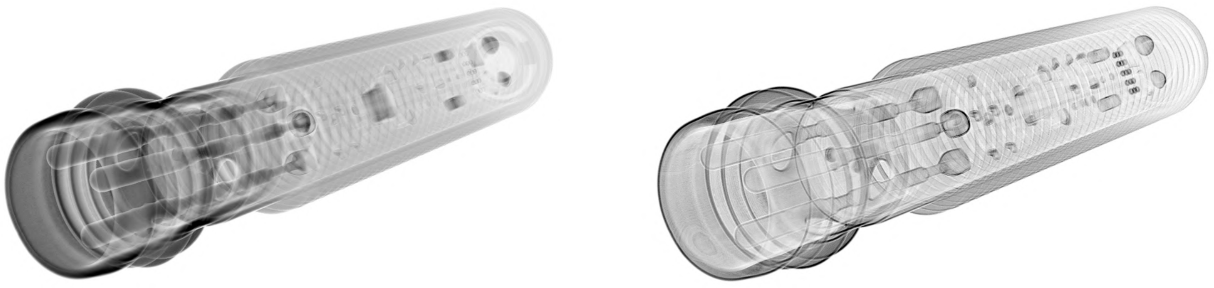


Fig. 3.6: This figure shows the rendering of a volume with mean attenuation rendering. The volume is a CT scan of a proximity sensor. The left image is the rendering without gradient rendering enabled, and the right image is the rendering with gradient rendering enabled. Gradient rendering is not very useful for this rendering mode. The colors of both images were inverted in order to remove the original black background.

3.2.5 "X-Ray" rendering

The idea behind x-ray rendering is to emulate the way x-rays interact with the volume. The principle is to make each cast rays act as if they were x-rays.

According to Beer's law, the formula to compute the attenuation of an x-ray through a multi material volume is the following:

$$I = I_0 \cdot e^{\sum_i -\mu_i \cdot d_i} \tag{3.4}$$

where I is the intensity of the ray, I_0 is the initial intensity of the ray, μ_i is the material's linear attenuation coefficient of material i and d_i is the distance that the ray traverses in the material i.

Here, this exponential formula is simplified by a linear regression to simplify the computation and is given by the following formula:

$$I = \prod_i (I_0 - \mu_i * step * \alpha) \tag{3.5}$$

where I is the intensity of the ray, I_0 is the initial intensity of the ray, μ_i is the attenuation value of voxel i, *step* is the current ray step and α is a constant that is used to scale the attenuation of the ray to allow users to modify the rendering result.

For our implementation, each ray were launched with an intensity value initiated to 65535 which is the maximum value that a voxel can have. Then, each time the ray hits a voxel, that value is decremented by the value of the voxel multiplied by both α and the ray step. For a better rendering result, the conversion of the attenuation value to the rendered color value is inverted, in order for rays that traverse lots of material to be rendered with a color close to white and rays that traverse little material to be rendered with a color close to black. This is done to be consistent with the other rendering modes.

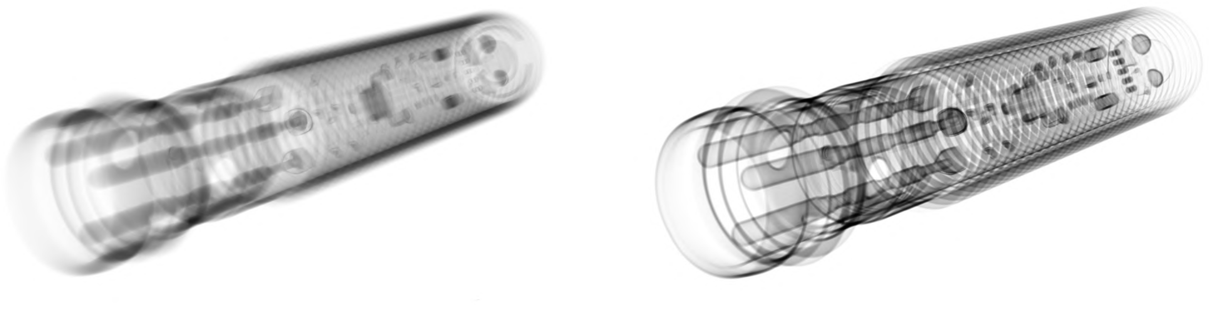


Fig. 3.7: This figure shows the rendering of a volume with x-ray rendering. The volume is a CT scan of a proximity sensor. The left image is the rendering without gradient rendering enabled, and the right image is the rendering with gradient rendering enabled. Here, gradient rendering is very useful since it gives a result that greatly amplifies the different parts of the volume. The colors of both images were inverted in order to remove the original black background.

3.3 Performance impact of each rendering mode

In this section, an analysis of the performance impact of each rendering mode will be presented. The performance impact of each rendering mode will be compared to the default rendering mode, as it is the simplest rendering mode and the one that was implemented first. The performance of rendering will be measured in frames per second (FPS). Since a lot of factors can influence the performance of the rendering, the only parameter that will be changed between the different rendering modes results is the rendering mode itself. The position of the camera, the volume, the thresholds and the brick size will be kept the same between the different rendering modes.

3.3.1 Gradient rendering

Since the gradient of bricks needs to be computed before being transferred from the RAM to the VRAM, gradient rendering adds transfer latency but shouldn't have a direct impact on the rendering performances. Indeed, once the bricks are loaded into the VRAM, the gradient values are already computed and as far as the pipeline is concerned, the gradient bricks in the pool are the bricks of the octree. Thus, gradient rendering only adds a small amount of processing time when new bricks are loaded into the VRAM, which in practice leads to a higher latency for bricks of the optimal level of detail to be shown on the screen.

But there is an indirect impact on the rendering performances. Indeed, since gradient rendering is used to amplify the borders between different materials, bricks that are mainly composed of the same value will become transparent and will not be rendered. This means that rays will traverse more bricks before being stopped by the opacity threshold, which will lead to a rendering performance impact that is proportional to the number of bricks that became transparent because of the gradient rendering.

In our tests, the performance decrease could be anywhere from 12% to 72% depending on the volume and the camera position. Which is a quite important performance impact. The impact could be reduced by changing the ray step value or the opacity threshold value. Indeed, in the case of gradient rendering, the artifacts that could appear because of those changes are not as important as in the default rendering mode since the artifacts are mostly pronounced on surfaces which are less pronounced when gradient rendering is enabled.

3.3.2 Isosurface rendering

Isosurface rendering has two characteristics that have an impact on the rendering performances. The first one is that all rays will stop at the first voxel they hit. This reduces considerably the number of bricks that are traversed by the rays, and thus the number of samples taken by the rays. This characteristic has a positive impact on the rendering performances since the rays will stop earlier leading to a faster render time.

The second characteristic is the diffuse lighting. The diffuse lighting is computed for each voxel that the ray hits. This means that both the normal vector of the voxel and the angle between that vector and the direction of the ray must be computed. This is done for each voxel that the ray hits, and thus the diffuse lighting has a negative impact on the rendering performances.

In practice, isosurface rendering has a positive impact on the rendering performance compared to the default rendering mode. Indeed, in our tests, switching from the default rendering mode to the isosurface rendering mode lead to a performance increase from 25% to 80% depending on the volume and the camera position.

3.3.3 Max/mean attenuation rendering

Since the only difference between both min and max attenuation rendering is the final value assigned to the ray, the performance impact of both rendering modes should be the same. Both rendering modes remove early ray stopping, and thus the rays will always traverse the whole volume. This leads to a performance impact proportional to the number of bricks that were skipped thanks to the early ray stopping.

In practice, the performance impact of max/mean attenuation rendering is quite important. In our tests, switching from the default rendering mode to the max/mean attenuation rendering mode led to a performance decrease of from 30% to 70% depending on the volume and the camera position.

3.3.4 X-ray rendering

X-ray rendering has a performance impact that is quite similar to the max/mean attenuation rendering. Indeed, the rays will always traverse the whole volume and thus the performance impact is proportional to the number of bricks that were skipped thanks to the early ray stopping. The difference between x-ray rendering and max/mean attenuation rendering is that in x-ray rendering, there can be early ray stopping if the ray's intensity value hits 0. This means that in some cases, the performance impact of x-ray rendering can be less than the performance impact of max/mean attenuation rendering.

Something to note about x-ray rendering is that the ray step value doesn't have a big impact on the rendering results. Thus, to compensate for the performance impact of x-ray rendering, the ray step value can be increased to reduce the number of samples taken by the rays without a big impact on the rendering results compared to the default rendering mode where increasing the ray step value would lead to the apparition of artifacts. This remark is also valid for the max/mean attenuation rendering modes.

In practice, the performance impact of x-ray rendering is quite important. Our tests showed that switching from the default rendering mode to the x-ray rendering mode led to a performance decrease of 36% to 75% depending on the volume and position at which the volume is observed from which is a bigger impact than max/mean attenuation rendering.

3.4 Conclusion

In this chapter, the multiple rendering modes that were added to the library were presented. All modes have their use in the context of non-destructive testing and metrology, and it was shown that the bricking and caching system didn't restrict the implementation of those rendering modes.

The performance impact of each rendering mode was also analyzed and compared to the default rendering mode. The performance impact, while large for some modes, can be reduced by increasing the ray step value to decrease the number of samples taken by the rays. This is especially useful for the gradient/x-ray/max/mean attenuation rendering modes, since the artifacts that could appear because of the increase of the ray step value are close to unnoticeable in those cases.

In Chapter 4, a global analysis of the performances of the library will be performed. Tests will be made in order to see the impact of the brick size on multiple aspects of the library, and the performance of the library on a real-world dataset will be analyzed. The results of those tests will be used to conclude on the optimal brick size for the library and to see if the library is in a state where it can be used by the Maestro CT software. An analysis of the impact of the ratio between the size of the volume and the size of the VRAM will also be performed to see the limits of the solution. Finally, the performance of the library on a real volume of 58.6 GB will be analyzed to see how large volumes are handled by it.

Chapter 4

Benchmarks and rendering results

In this chapter, benchmarks and rendering results will be presented. The first section will present the different volumes that were used for the benchmarks. The second section will analyze the impact of the size of the bricks on the octree generation time, the file size, the rendering performances and the memory usage. The third section will analyze the impact of the size of the cache pools on the rendering. And the last section will present two real examples with one showing the usefulness of our solution compared to down-sampling and the other showing the rendering of a volume that is a lot bigger than the VRAM.

4.1 Presentation of the datasets

Two groups of volumes will be used for the testing of the performances of the software. The voxel sizes and file sizes of all volumes in each group are reported in Table 4.1.

Volume Group	**Voxel sizes (file size)**
1	$840 * 1103 * 840$ (1.5 GB), $1401 * 1838 * 1401$ (6.8 GB), $1681 * 2206 * 1681$ (11.8 GB), $1867 * 2451 * 1867$ (16.2 GB)
2	$850 * 3675 * 850$ (5 GB), $1000 * 5000 * 1000$ (10 GB), $1250 * 5500 * 1250$ (16 GB), $1500 * 6000 * 1500$ (26 GB)

Tab. 4.1: This table shows the different volumes for each group that will be used for the benchmarks. The voxel sizes, and the file sizes are shown for each volume.

Figure 4.1 shows the different volumes that will be used for the benchmarks. The volumes of the first group are CT scans of a proximity sensor and the volumes of the second group are simulated CT scans of a cylinder containing spherical marbles arragned in a helical pattern.

4.2 Study of the global impact of the chosen size of bricks

In this section, we will analyze the impact of the sizes of the bricks on the file size, the rendering performances, the memory usage and the octree generation time. We'll use the datasets presented in Section 4.1 to perform these tests. The brick size analyzed will be of the size 8, 16, 32 and 64.

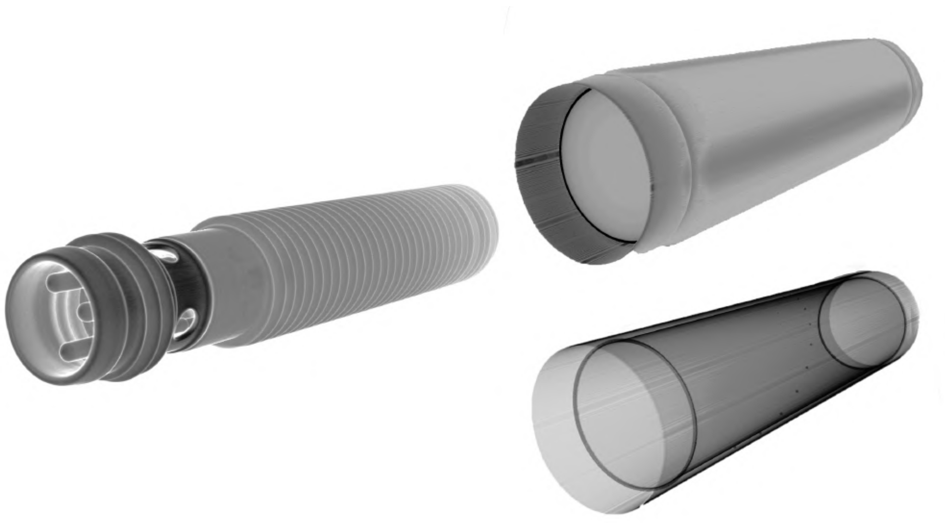


Fig. 4.1: The left part of the image shows the volumes of the first group, and the right part of the image shows the volumes of the second group. The volumes of the first group are CT scans of a proximity sensor and the volumes of the second group are a simulated CT scan of a volume that represent a cylinder containing small marbles. In the right part, the default rendering mode is shown on the top and the gradient x-ray rendering mode is shown on the bottom. As can be seen on the bottom part of the image, small marbles are present around the volume. These spherical marbles are arranged according to a helical pattern, with a linearly decreasing radius. Only one volume is shown per group, since the only difference between the volumes of the same group is their sizes. The images have their color inverted in order to remove the original black background.

4.2.1 Octree generation time

The octree generation time is an unavoidable process that adds processing time the first time the volume is opened in the GigaVoxel library. It is thus imperative to make this process as fast as possible, especially for large volumes. The process was described in details in Section 2.3.3.

One of the biggest impacts of the brick size on the octree generation time is the number of levels present in it. Indeed, the more levels the octree has, the more MIP-map levels needs to be computed at the generation step. Also, for each level, a node file, a brick file and a range file need to be written to the disk. The number of levels composing an octree is given by the following formula:

$$\text{levels} = \left\lceil \log_2 \left(\frac{\text{volume voxel size}}{\text{brick size}} \right) \right\rceil \tag{4.1}$$

As can be determined by the formula, the smaller the bricks, the more levels the octree will have for volumes of the same size.

The other parameter that has a big impact on the octree generation time and that is linked to the brick size is the number of bricks themselves. As stated in Section 2.2.1, all bricks have a border of redundant data around them that is composed of voxels from the neighbor bricks. These redundant voxels are used for interpolation and are necessary to have a good rendering. Since the second step of the octree generation at each level is to compute the border of the bricks, the more bricks there are, the more time is needed to compute them. The number of border voxels is also dependent on the brick size. Smaller bricks have proportionally more voxels in their borders, and thus the border computation step takes more time, since more voxels need to be copied from the neighbors of the bricks. Table 4.2 shows the number and proportion of border voxels for all tested brick sizes. As can be seen in the table, the proportion of border voxels decreases as the size of the bricks increases.

Brick size	**8**	**16**	**32**	**64**
Number of data voxels	512	4096	32768	262144
Number of border voxels	488	1736	6536	25352
Proportion of redundant data	48.8%	29.8%	16.6%	8.82%

Tab. 4.2: This table shows both the number and proportion of border voxels for each considered brick sizes. As can be seen, the proportion of voxels that are used for the border decreases as the size of the bricks increases.

The way data is read from the disk is also linked to the brick size. Indeed, since the volume is read by slices of bricks, the larger the brick, the more is read at one. If bricks are too small, the octree generation step might be slower because the DataStructureIOManager object requests a small quantity of data too frequently and if the brick is too big, the generation step might be slower because it reads the volume with too big chunks at a time.

Results

Table 4.3 shows the measured octree generation time for all volumes at all brick sizes. These tests were only made on machine B since it possesses 128 GB of RAM and thus avoids any kind of swapping. Indeed, during our tests on machine A, we quickly noticed swapping during the generation step even though the machine has 32 GB of RAM, which is larger than the size of any volumes in the dataset. The issue originates from the MIP-map algorithm. Indeed, when generating a level, the lower level data needs to be in memory. Thus, in the worst case, both the lowest level of data and the second-lowest level of data are in memory at the same time (when the generation of the second-lowest level is close to be finished), which can represent a large amount of data in memory. The swapping only added time to the generation step, as it will eventually return with the correct output.

As can be seen, the generation time decreases as the size of the brick increases. If we analyze the results, we notice quite quickly that the generation time is close to linear with the size of the volume for volumes of the same group. The difference between the generation time in each group is due to the fact that the volumes have different voxel size and different number of constant bricks. The different sizes can cause the generation time to be different, since the number of levels of the octree is determined by the largest composing coordinate of the volume. The number of constant bricks can also have an impact on the generation time since at the generation, if a brick is only composed of voxels of the same value, the algorithm will skip the step of writing the brick to the file to reduce redundancy.

Volume 1\Brick size	**8**	**16**	**32**	**64**
1.5 GB	017s	015s	014s	014s
6.8 GB	076s	066s	062s	063s
11.8 GB	136s	116s	108s	109s
16.2 GB	185s	159s	147s	149s
Volume 2\Brick size	**8**	**16**	**32**	**64**
4.9 GB	081s	070s	064s	064s
9.3 GB	142s	108s	091s	085s
16.0 GB	297s	242s	212s	205s
25.1 GB	441s	365s	330s	320s

Tab. 4.3: This table shows the octree generation time for all volumes in the first and second group for different brick sizes. All the times were measured on machine B and are expressed in seconds.

4.2.2 File size

There are multiple impacts that the brick size has on the file size, and the first is the number of levels in an octree. Indeed, the level of octree directly impacts the file size, since each level needs to have its own files, which adds overhead. Each level also represents a different level of detail of the volume that needs to be stored and the number of level of resolution that an octree has is given by the Formula 4.1.

The second impact that the brick size has on the file size is the number of bricks that are needed to store the volume. Indeed, each brick needs to have a node in the node file which indicates if the brick is empty or not and where the data of the brick is stored in the brick file. Each node record requires 4 bytes of storage (1 32-bit value) as was told in Section 2.2.1. Each brick also has a range value associated to it which are 2 16-bit values (2 unsigned short) as was described in Section 2.3.4. Thus, the more bricks there are, the more overhead is added to the node and range files. The number of bricks obviously depends on the size of the bricks, since the bigger the bricks are, the fewer bricks are needed to store the volume. But it also depends on the size of the volume and the number of empty/constant bricks in the volume. Indeed, bricks that are either empty or composed of voxels of the same value are not stored in the brick file to reduce redundancy. Thus, the more empty/constant bricks there are in the volume, the fewer bricks are stored in the brick file. But the bigger a brick is, the less probability there is for a brick to be only composed of voxels of the same value.

The third impact of the brick size on the file size is the overhead used by the borders in each brick. Since this border is always a voxel thick no matter the brick size, the bigger the bricks are, the more voxels are needed for the border. But while more voxels are required for the border, the proportion of voxels in a brick that are used for the border decreases as the size of the bricks increases. This can be seen in Table 4.2.

Results

To test the impact of the brick size on the file size, the volumes of both groups were used. All volumes were generated with the 4 brick sizes and the amounts of storage required to store all the node, brick and range files of each of the volumes were measured and are shown in Table 4.4.

Volume 1 \Brick size	**8**	**16**	**32**	**64**
1.5 GB	2.37 (+ 58%)	1.75 (+ 17%)	1.58 (+ 5%)	1.64 (+ 9%)
6.8 GB	10.6 (+ 55%)	7.86 (+ 16%)	6.88 (+ 1%)	6.80 (+ 0%)
11.8 GB	18.7 (+ 58%)	13.5 (+ 14%)	11.7 (- 1%)	11.6 (- 2%)
16.2 GB	25.4 (+ 56%)	18.4 (+ 14%)	16.0 (- 1%)	15.6 (- 4%)
Volume 2 \Brick size	**8**	**16**	**32**	**64**
4.9 GB	11.8 (+ 141%)	8.4 (+ 71%)	7.1 (+ 44%)	7.0 (+ 43%)
9.3 GB	18.2 (+ 95%)	10.7 (+ 15%)	8.7 (- 6%)	8.2 (- 12%)
16.0 GB	40.7 (+ 154%)	22.7 (+ 41%)	23.0 (+ 44%)	21.0 (+ 31%)
25.1 GB	61.1 (+ 143%)	41.7 (+ 66%)	35.0 (+ 39%)	33.1 (+ 32%)

Tab. 4.4: This table shows the file size of the volumes of both the first group and the second group for the different brick sizes. The file size is shown in GB, and the percentage of increase or decrease compared to the initial raw file size is shown in parentheses.

For the volumes of the first group, the file size decreases as the size of the bricks increases. This is due to the fact that for lower brick sizes, more octree levels are required, and more border voxels are needed, as can be seen in Table 4.2. For bigger brick files, it is possible to see a decrease in the file size compared to the original .raw file. This is because, in octrees, if a brick is only composed of voxels of the same value, the brick is not stored in the file to reduce redundancy. This effect is present for all brick sizes and should be more present for smaller brick sizes, since the probability for a brick to be only composed of voxels of the same value decreases as the size of the bricks increases. But for smaller brick sizes, this effect is overshadowed by the increase in the number of levels of the octree and the number of border voxels that are needed.

For the volumes of the second group, we still see that bigger bricks use less space than smaller brick, but there is a strong increase in the overall size of the files. This is because the volumes of the second group are simulated and do not come from a real CT scan of a volume. Because of the way the CT scan was simulated, the volume has not a lot of empty voxels and the volume is a lot more long than large. This causes the number of levels of the octree to be higher compared to volumes of similar sizes in the first group, since the level of the octree is determined by the largest composing coordinate of the volume. In practice, the behavior of real CT scanned volumes should be closer to the one of the first group.

4.2.3 Rendering performances and memory usage

Because both rendering performance and memory usage are dependent on a lot of parameters that are hard to control, like the position in 3D space and the zoom level, we haven't done an experimental analysis of the subject since there is no easy way to compare the performances of multiple brick size configurations fairly. But, we can still analyze the question from a theoretical angle.

Rendering performances

For the rendering performances, the ray step, the level of detail selection and the Provider's influence on performance depend on the brick size.

Indeed, for bricks that are far enough, the ray step is dependent on the size of the brick and on the level of detail it is traversing, as was discussed in Section 2.3.5. While the ray step's value is dependent on the brick size, it is actually used to make the ray step stay at the same size no matter the brick size used. But since there are fewer levels of details and that the difference between multiple levels of details of the octree is greater, there will be fewer changes in the overall level of detail used in the rendering. The impact that this has on the ray step is that its value will stay longer at the same value before increasing, since, in theory, the value only changes when passing through a brick of a different level of detail. This leads to more sampling and more time required for rendering.

For the Provider, the potential loss in rendering performance stands from the fact that the Provider can only filter out bricks. This means that the bigger the bricks, the less precise the filtering made by the Provider can be. Thus, for a same volume and a same threshold, more bricks will be filtered out by the Provider on the volume with a smaller brick size than on the volume on a bigger brick size. The more empty bricks there are, the fewer samples will be taken during rendering, which will reduce the required time to render thanks to empty space skipping.

One positive factor of bigger bricks in terms of performance is the time needed to march the octree in order to find the brick that we want to render. Indeed, since the larger the bricks are, the fewer bricks are used for rendering, fewer bricks need to be fetched for each frame in the cache pools. Also, since bigger bricks lead to fewer levels of details in the octree, less time is needed to fetch bricks in lower levels of details. This reduces the time necessary to fetch bricks in pools, and thus should increase performances.

In practice, we didn't notice a big difference in the rendering performances between the different brick sizes. But as already stated, because of the way the rendering is done, it is hard to compare the performances of different brick sizes fairly.

Memory usage

For the memory usage, the bigger the bricks, the less level of detail there is and the less can be filtered out by the culling.

The number of available level of detail has a direct impact on the memory usage because of the bigger difference between each level of detail of the brick. Indeed, since fewer levels of details are available, bigger bricks will be used when otherwise smaller one could have been used if more levels of details were available. This leads to more memory being used in most rendering scenarios since fewer levels of detail are available.

Since bricks are bigger and that the culling can only filter out bricks, a lot of voxels that are not visible because they are outside the screen will still take space in the cache pool as at least one voxel inside the brick is visible on the screen. Fewer voxels would be used if smaller bricks were used since the culling could more precisely filter out unseen voxels.

4.2.4 Conclusion

Since brick size cannot be easily changed without recompilation, we decided to analyze brick sizes in details in order to see if a single brick size could be used for most if not all situations. Based on the analysis done in the previous sections, we think that the brick size of $32 * 32 * 32$ voxels is the best size among the one tested. It offers good file sizes and performance during our testing. This size might be a bit big for volumes smaller than 2 GB, but since we're aiming at enabling the rendering of volumes larger than the VRAM, these volumes shouldn't be used with this technique in the first place. This size might become too small for very large volumes (maybe for volumes bigger than 100 GB) but the only issue would be file sizes and maybe generation time.

4.3 Study of the impact of the size of cache pools on rendering

In this section, we will analyze the impact of the size of the cache pools on the rendering. As in the last section, we'll use the datasets presented in Section 4.1 to perform these tests and on the machines presented in Appendix A. Since the cache pool is a parameter that is defined in the code, we can simulate the impact of lower VRAM sizes by reducing the size of the cache pools without having to change the hardware. This allows us to see what happens when the pool is too small for the volume that is being rendered.

We won't analyze the impact of a small node cache pool since, in practice, we never had it close to being full during our tests where the pool was set to only 64 MB. The node cache pool is used to store the nodes of the octree. Since the nodes are only 4 bytes long, the pool, in its current size, can store approximately 16 million nodes, which is more than enough for pretty much any volume derived from CT scans.

If all bricks in the cache are currently used for rendering, the cache won't be able to serve requests to load new ones. For those that can't be loaded in the cache, the octree traversal will thus automatically stop at the closest parent brick that is in the cache. This will lead to the rendering of bricks that are not at an optimal level of detail, which leads to parts of the volume being rendered at a lower level of detail than they should be, making the rendering less precise. But if the user changes the point of view, some bricks will be marked as unused and will be replaced by new bricks that are needed for the new point of view. This will lead to the rendering of the new bricks at the correct level of detail if this change of point of view causes enough bricks to be marked as unused. Otherwise, the rendering will still be less precise, but probably not for the same bricks as before.

The fact that if the cache is full, bricks with a lower level of detail are loaded might actually lead to an increase in performance compared to the same situation with a bigger cache pool. Indeed, since the ray step is dependent on the level of detail of the brick, the lower the level of detail the brick has, the fewer samples are taken during rendering. This leads to a decrease in the time needed to render the volume.

During our testing, we found that if the Provider threshold was used to filter out bricks correctly (meaning setting it to the same threshold as the shader), the cache pool could be set to a ratio as low as 1/10 before having a noticeable impact on the rendering. This is because the Provider can filter out a lot of bricks that are not visible on the screen, and thus the cache pool is less likely to be full. We also wanted to note that if the user does not use any kind of filtering, the cache pool is unlikely to be full. Since the air voxels would hide a lot of the volume that would not be loaded into the cache thanks to culling.

4.4 Test on a real use-case

In this section, we will present two real examples that demonstrate the usefulness of our solution in the context of non-destructive analysis of mechanical parts. The first will show and compare volume of different sizes that are down-sampled from an initial large volume to show why down-sampling is not an ideal solution to allow the rendering of volumes larger than the VRAM and the second example will show the rendering of a volume that is a lot bigger than the VRAM.

4.4.1 Comparison with down-sampling

For this section, we will compare the visibility of the marbles in the second group of volumes from the dataset. The marbles are located all around a cylinder, and their size decreases as they approach the top of the cylinder. Because of this, the more down sampling is done, the less the higher marbles will be visible.

To prove the usefulness of our method, we will compare the volume of 5.1 GB and the volume of 26.3 GB. The volume of 5.1 GB will emulate what a user with a graphics card with 6 GB of VRAM would be able to see when trying to render the 26.3 volume with down-sampling. We will compare how many marbles are visible between the two render and thus show that our solution allows a user to see all the details of a volume even when his VRAM is multiple time smaller than the size of the volume. To show that this result would work with a graphics card of 6 GB (which we don't have), we will instead limit the size of our cache pool, since it is what would happen if we had only 6 GB of VRAM.

On the volume of 5 GB, only 17 marbles could be seen with the 18th one being barely visible. When doing the same experiment on the volume of 26.3 GB, 18 marbles can be clearly seen with the 19th one being barely visible.

As said above, more details were made visible thanks to our solution and thanks to the level of detail selection and culling mechanism. A hypothetical user limited by his hardware was able to render the volume with maximum details, whereas using solutions like subsampling would have made fewer details visible and other solutions like splitting the volume into multiple smaller parts would have made the analyzing process annoying.

4.4.2 Real large volume

For this section, a volume that wasn't presented in Section 4.1 will be tested. The volume has a size of $3091 * 3091 * 3298$ voxels, which represent 58.6 GB of file storage. It is a CT scan of the X-RIS logo made with Legos and can be seen on figure 4.2. Because of the large size of the volume, this test was only made on machine B (See Annex A) thanks to its large amount of RAM that could fit the volume. All results were measured with a brick of size 32 as it was concluded to be the optimal brick size in Section 4.2.4.

The octree generation process took 504s and the final size of the octree was of 52.2 GB, which is a reduction of 11%. This large reduction in required storage size can be explained by the large amount of air present in the volume compared to the volumes presented in Section 4.1. Indeed, since voxel values of 0 are not stored in the octree, having large amount of air in a volume will lead to an impactful reduction in its octree size. The generation time took is in the expected range when compared to volumes of the

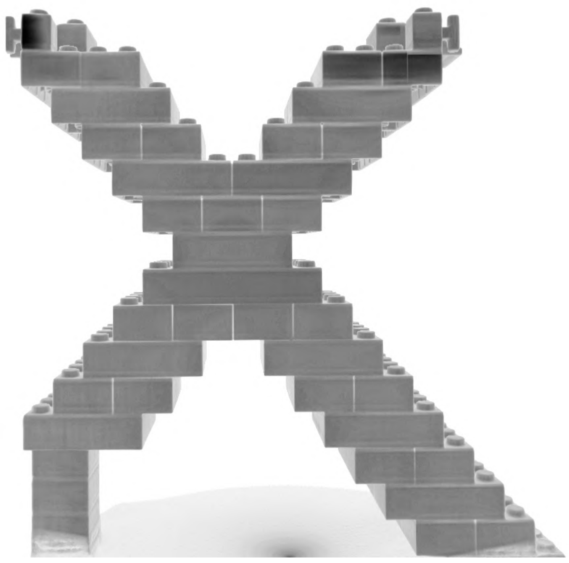


Fig. 4.2: This figure shows a rendering of a CT scan of the logo of X-RIS constructed from Legos. This scan is composed of $3091 * 3091 * 3298$ voxels, which represent 58.6 GB of file storage. The colors of this image were inverted in order to remove the original black background.

first group.

The performance of the rendering was very satisfactory (30-60 FPS) in the default rendering mode, even when viewing the volume from a small distance and with a smaller ray step than the default one. The performance when using the max/mean/x-ray rendering modes was worse (15-25 FPS), as was expected from the conclusion presented in Section 3.4. But with a reduced ray step, the performance in these rendering modes greatly increased to a satisfactory range (35-45 FPS).

For the memory consumption, since the brick pool is set to be 70% of the available VRAM when launching the program, approximately 16.5 GB of VRAM were reserved for the brick pool. During our testing, the proportion of the brick pool used during a single frame didn't go past 10% when using the default rendering mode and never past 35% when using rendering modes that do not use early ray stopping.

4.4.3 Comment on general performances

Something that was noticed during our testing is that the software is hard to run, especially in highly transparent scenarios and at a close zoom level on both machine A and B, which are high-end machines. Since the software is meant to be used as a tool for analysis, users don't need to have a high frame rate and as such, the software, while bring hard to run, has acceptable performance. But, performance is something that we think could be improved in the future.

In the worst cases, the frame rate can drop to single digit values when the volume is very transparent and the user is visualizing the volume with a large zoom. In these cases, using the Provider threshold to filter out bricks that are not visible on the screen and increasing the size of the ray step can help to increase the frame rate to a more acceptable value in the 25-35 FPS range. But in some cases, increasing the ray step may lead to wood grain artifacts as explained is Section 2.3.5.

This is why a focus was made on the implementation of features that help with the performance of the software like the range file implementation, the selection of the size of the ray step and the option at the octree generation that allows the user to remove outer voxels based on a radius. The user can also limit the deepest level of detail loaded by the software to increase performances.

4.5 Conclusion

In this chapter, the impact of the brick size on both the file size and on the octree generation time were analyzed. This analysis lead to the conclusion that using bricks of size 32 is the best size for most, if not all, situations thanks to its balance between small file size and octree generation time and its performance as the smallest manageable unit in the pipeline.

We then analyzed the impact of the size of the cache pools on the rendering and showed that the cache pool could be set to a ratio of at least 1/10 of the size of the volume without having a noticeable impact on the rendering. Below that ratio, some bricks of the volume may not load at their optimal level of detail and stutters may appear when the user change the view at which the volume is rendered which makes the program feel less performant. Those issues get amplified as the ratio between the size of the brick pool and the quantity of data that should be seen at the optimal level decrease. These issues themselves to not make the program unusable but greatly decrease the user experience as their presence increase.

Two example of rendering results were then presented. The first showed the advantages of our solution compared to down-sampling and the second analyzed the performance of our solution on a large volume of 58.6 GB. This analysis showed that our solution with a brick size of 32 had satisfactory rendering performance and an acceptable octree generation time.

Chapter 5

Surface extraction of voxel volumes

The goal of this chapter is to analyze the different available algorithms for the extraction of a surface from a voxel volume. The principle behind surface extraction is to generate a mesh with a sub-voxelic precision that represents the surface of a volume given an isosurface value and voxel volume. Having a sub-voxelic precision means that our mesh definition shouldn't be restricted by the voxel grid of the initial volume, and should have a precision smaller than the size of a voxel. The isosurface value is a value used to determine what voxels of the initial volume are considered as part of the surface that the user wants to extract. The use of surface extraction in the context of metrology analysis of CT data is to ease the measurement of the features of a scanned object.

This chapter is divided into 3 sections. The first section will present the state of the art of surface extraction techniques. The second section will present the pipeline that was developed to test the different algorithms, and the last section will present the results of the tests that were made on the different algorithms. Finally, a discussion of the results will serve as a conclusion to this chapter.

5.1 State of the art of surface extraction techniques

In this section, the state of the art of surface extraction techniques in the context of 3D CT data will be presented.

One of the first and most well known surface extraction techniques is called marching cubes and was introduced by Lorensen et al. [12]. The goal of the marching cubes algorithm is to create a triangle model of a constant density surface from a 3D voxel volume.

The principle is to march a logical cube composed of 8 voxels along the volume and to determine how the surface intersects the cube. To do so, a value must be defined by the user representing the intensity value of the surface. The 8 voxels inside the cube will then be considered as either inside the surface if their intensity value is higher or equal than the user defined value, or outside the surface if their intensity value is lower than the user defined value. Depending on the status (inside or outside) of the 8 voxels, a unique configuration number is assigned to the cube. This configuration number is thus 8 bits long, where each bit of the number represents the status of a voxel (again either inside or outside the surface). There are thus $2^8 = 256$ possible configurations. Figure 5.1 shows an example step of the marching cubes algorithm.

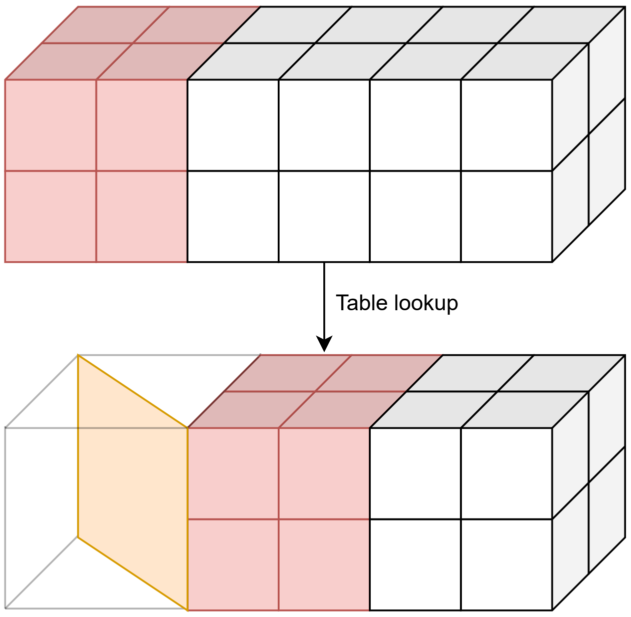


Fig. 5.1: Example step of the marching cubes algorithm. The cube is composed of 8 voxels and the surface is represented by the orange triangle mesh. The red voxels represent the current position of the logical marching cube.

A table is then used to map each possible configuration of the cube to a triangular surface. This triangular surface determines the approximation of the surface inside the cube. Having 256 triangular configurations inside a single table would be redundant, since some configurations are symmetrical to others.

The first symmetric reduction is to consider cubes where the status of the voxels are inverted as the same configuration. This is because since the goal is to determine the border between two mediums (inside and outside), two volumes with inverted status of their voxels would have the same surface. This reduces the number of configurations to 128. The second reduction is to consider rotations of cubes as the same configuration. This reduces the number of configurations to 15. Those 15 configurations are shown in Figure 5.2.

Once all the cubes have been processed, linear interpolation is used to determine the exact position of the vertices of the triangles. The final result is a triangle mesh that represents the surface of the volume. The normal of each triangle is also computed to allow the rendering of the mesh. This is done by computing the gradient of the surface thanks to a central difference scheme.

Yagüe-Fabra et al. [13] proposed a new 3D surface extraction method with sub-voxel precision based on the application of the Canny filter in 3 dimensions.

Their algorithm is composed of 4 phases. The first phase is a preliminary surface detection. A Gaussian filter will be applied to the 3D image data in order to reduce the

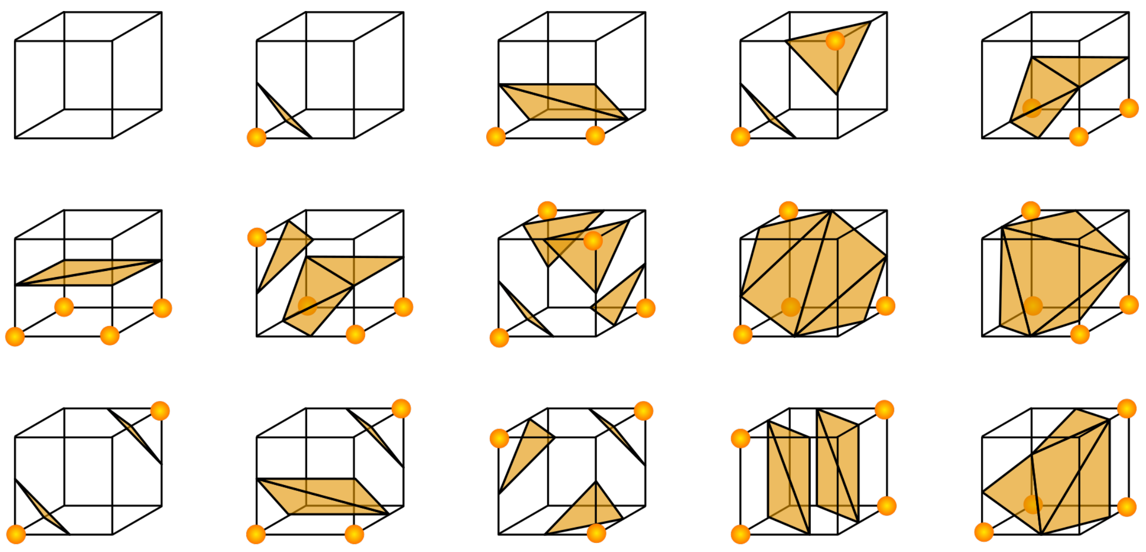


Fig. 5.2: This figure[1] shows the 15 possible configurations of cubes in the marching cubes algorithm. Orange voxels are inside the surface and "empty" voxels are outside the surface. The orange triangle mesh is the mesh that should be drawn to represent the surface inside the cube.

image noise. Afterward, the Canny operator will be applied in the X-Y plane, the X-Z plane and the Y-Z plane.

The Canny edge detection algorithm is composed of multiple steps. The first step is to remove the noise thanks to a Gaussian filter. Then the second order directional derivative is computed and non-maximum suppression is applied to it. The purpose of the non-maximum suppression is to make all the detected edges thin. The principle of this step is to find the maximum pixel value at an edge and to remove the pixels around that maximum value pixel in both gradient directions. The last step of the Canny edge detection algorithm is to apply a double threshold to the image. The principle of double threshold is to define a strong and weak threshold. Each value above the strong threshold will be kept, all values below the weak threshold will be discarded, and a value between the two threshold will be considered as strong only if the pixel is in contact with a strong pixel. All pixels that aren't strong will be discarded. These Canny computations will be done for each slice in each of the 3 planes, resulting in 3 3D images where Canny is applied.

The second step is the sub-voxel refinement. The principle of this step is to try to determine where all points of the surface are on the model based on the voxel volume while having sub-voxel precision. This means that the points of the surface that is being searched for can be represented anywhere in 3D dimension without being limited by the voxel grid. The next step is to apply a center of gravity algorithm to the gradient image at each point defined by the Canny filter. The center of gravity will be considered as the point of the surface.

The result is a point cloud that defines the surface that is being searched. During their

[1]This figure was made by "Ryoshoru" and was taken from `https://commons.wikimedia.org/wiki/File:MarchingCubesEdit.svg`. It is under the Creative Commons Attribution-Share Alike 4.0 International license and has not been modified.

testing, Yagüe-Fabra et al. [13] found that their surface reconstruction from the cloud of points would always have a predictable and systematic error value on their results. They thus decided to add an error correcting step that would compensate for that error that was always in the positive direction to the normal of the surface, leading to reconstructed volumes that were thicker than expected. They also added a step that would apply a correction of the bias given the real measurement of a particular feature.

Yagüe-Fabra et al. [13] then showed that their pipeline had a reduction in measurement uncertainty of up to 60% compared to the local threshold technique.

5.2 Introduction

There are two types of surface extraction algorithms. The first type will apply a pipeline of filters to the initial voxel volume, which will lead to a point cloud representation of its surface. Then, the precision of that point cloud will be improved by a sub-voxel refinement step and the final step will determine the surface represented by the point cloud.

The second type of surface extraction algorithms will directly determine the surface represented by the voxel volume without using any secondary representation. It will directly use the input voxel data. Figure 5.3 shows a representation of both type of pipelines.

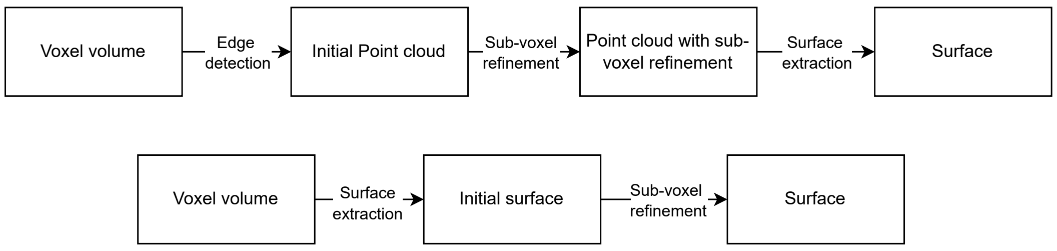


Fig. 5.3: This figure shows a representation of the two type of surface extraction pipelines. The top one is the pipeline based on point cloud and the bottom one represents the pipeline that uses standalone surface extraction algorithms.

One challenge of surface extraction is to deal with the noise and artifacts that are present in the voxel volume. In CT, these noises and artifacts are caused by the CT reconstruction process and can lead to a surface that is not coherent with the scanned object. An example of the noise and artifacts that can be present in the voxel volume is shown in Figure 5.4. One of the goals of the surface extraction algorithms is to be able to not be affected by these noise and artifacts while extracting the surface.

In the next sections, we will present the different algorithms that were studied for both types of surface extraction algorithms, compare their performances and present our test results.

5.3 Surface extraction pipeline

In this first section, we will present the first type of surface extraction algorithms. Our plan for this section was to first develop a generic pipeline that computes the point cloud on which the surface will be built. The point's positions will then be modified according

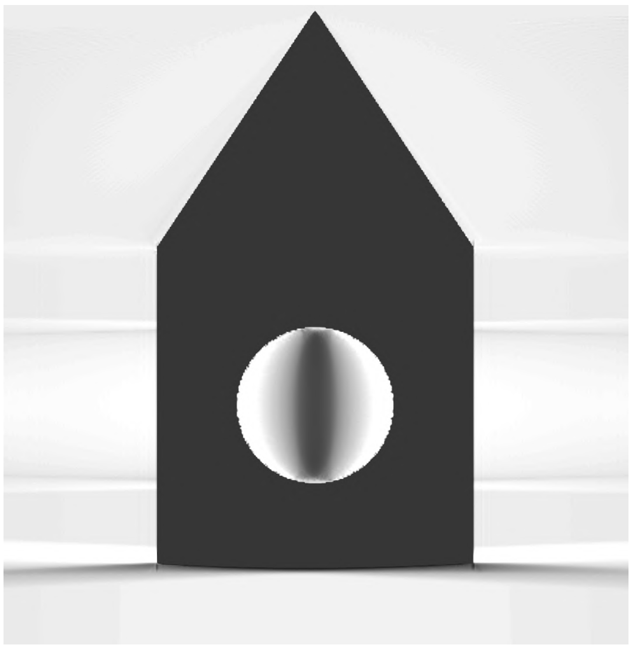


Fig. 5.4: This figure shows a slice of a voxel volume that was reconstructed by a simulation of a CT scan. The image shows the noise and artifacts that are present in the voxel volume, especially at the base of the cube, where the artifacts make it hard to distinguish the base of the cube. The artifacts are the grey beams that are present in the image.

to a sub-voxel refinement algorithm. The last step of the pipeline will be to compute the mesh that best represents the point cloud. The goal of this pipeline was to test multiple algorithms for each part of the pipeline and to determine which algorithms would give the best results. The pipeline was implemented in ITK and VTK.

The following subsections will present the major parts of the pipeline and will analyze all the algorithms that were studied for each part of the pipeline.

5.3.1 Edge detection

The first step of the pipeline is to detect the surface voxels of the CT volume thanks to edge detection. In order to do so, the isosurface value is used to detect the voxels that are part of the surface and in our case, we decided to use the Canny edge detection algorithm to detect the edges of the surface of the volume. We decided to use this algorithm because it was already implemented in ITK, was compatible with 3D voxel data and produced good results on the tests we made.

The Canny edge detection algorithm was described by [14]. The first step of the algorithm is to apply a Gaussian filter to the image in order to reduce the noise. The second step is to compute the direction and the magnitude of the gradient of the image. Gradient local maxima are first considered as edge candidates. The last step of the algorithm is to apply a hysteresis threshold, which will remove all edges that are not considered as strong. The principle is to define a low and a high threshold. Any point above the high threshold will be considered as being part of a strong edge, and any point below the low threshold will be considered as being par of a weak edge. Strong edges are kept in the final image and weak edges are removed. For the points that are between the two thresholds, they are considered as either strong or weak depending on if they are in contact with a strong or weak edge point.

The Canny algorithm was designed for 2D images, but it can be quite easily modified to support 3D voxel images, since all the algorithms used in the stages of the Canny algorithm have versions that work with 3D voxel data.

Before applying the edge detection algorithm, a filter was implemented in order to filter out unneeded values from the initial voxel volume set. We decided to apply the Otsu threshold algorithm described by Otsu [15] which will create a mask that separates the voxels representing the air inside the CT reconstruction volume from the ones that represent the scanned volume. That mask will then be applied to the original image before being given to the edge detection algorithm in order to reduce the noise of the initial data set.

The principle behind the Otsu threshold method is to find the best threshold that separates the voxels into two groups: the foreground and the background. To determine what is the best threshold, the method will compute the variance of the two groups and will try to minimize the variance between the two groups. The variance is computed by the formula:

$$\sigma^2(t) = w_0(t) \times \sigma_0^2(t) + w_1(t) \times \sigma_1^2(t) \tag{5.1}$$

where $w_0(t)$ and $w_1(t)$ are the weights of the two groups when the threshold is set to t (in our case, the proportion of voxels in the two groups) and $\sigma_0^2(t)$ and $\sigma_1^2(t)$ are the variances of the two groups when the threshold is set to t. The threshold that minimizes the variance is the threshold that will be used to separate the two groups.

We decided to implement this step of the pipeline in ITK since ITK already proposes an implementation of both the Canny edge detection algorithm and the Otsu threshold algorithm. It also proposes a multitude of other algorithms that can be used to filter the data before applying the edge detection algorithm, and easy ways to import and export the data.

From the output of the edge detection algorithm, we can extract a point cloud by taking the center of the voxels that are detected as being part of the surface. This point cloud is a first approximation of the surface, since the points are limited by the voxels they are taken from. The next step of the pipeline is to refine the position of the points in order to have a sub-voxelic precision.

5.3.2 Sub-Voxel refinement

The principle behind this step is to improve the precision of the point cloud that was obtained at the end of the last step, in order for these points to not be limited by the voxel grid. Indeed, right now and like said in the last section, points can only be placed at the center of voxels. The goal of this step is to remove that restriction on the position of the points. While we want to remove that restriction, we also don't want to move the points too far from their initial position, since it would lead to a representation of the surface that might not be coherent with the initial data.

To determine the new position of the points, we decided to use the gradient of the volume. The gradient is used to find the point of the surface because the gradient, by definition, is the direction of maximum variation of the voxels value. Thus, at a surface of a volume, the gradient will be a local maximum.

The idea is to sample the gradient of the volume in the direction of the gradient at each point of the point cloud. To determine the direction of the gradient, we will use the gradient direction of the voxel that is the closest to the point. Since the gradient of the image was already computed at the edge detection step, there is no need to recompute it. We then sample the gradient in that direction and move the point to the position where the gradient is the maximum. We obtain sub-voxelic precision by using an interpolation method to sample the gradient.

This process can be repeated multiple times and should, after some time, converge to a single value for each point. There are multiple parameters that can be tuned to either have a better precision or a faster convergence time. There is the step distance taken to sample the points in the gradient direction, the number of samples taken in the gradient direction and the number of iterations of the algorithm.

This part of the pipeline was partly implemented in ITK and partly in VTK. ITK was used to compute the gradient of the image, and VTK was used to manipulate the points of the point cloud.

5.3.3 Surface extraction

This step will generate a triangle mesh from the set of points obtained at the previous step. The idea is to use a surface reconstruction algorithm that will generate a mesh that represents the surface of the volume. There are multiple algorithms that can be used to do so. In the next sections, we will present the different algorithms that were studied and explain how they work.

Extract surface algorithm

This surface extraction algorithm is based on the paper of Curless et al. [16]. The algorithm uses a volumetric method to generate a surface from a multitude of range images. Range images are images that give the distance of each sample on the surface to the camera. The algorithm will then use the range images to generate a point cloud that represents the surface of the volume. That point cloud will then be converted to a volumetric representation of the surface of the volume. The algorithm will then use the volumetric representation of the surface to generate a mesh that represents the surface of the volume, thanks to the Flying Edges algorithm. Its results should be thus similar in some cases to the results of the Flying Edges algorithm.

The algorithm is already implemented in VTK, and it is the version we used for our tests.

Poisson surface reconstruction

The Poisson surface reconstruction algorithm was described by Kazhdan et al. [17]. The basic idea behind the Poisson surface reconstruction algorithm is to use the Poisson equation to reconstruct the surface of the volume from a set of points whose normal vectors are known. The idea is that the set of points with their known normal vectors can be seen as samples from the gradient of the model's indicator function, where the indicator function of a model is a function that is equal to 1 inside the model and 0 outside the model (which actually represent the volume). From that point of view, we want to find the indicator function of the model whose gradient is the closest to the vector field defined by the set of points and their normal vectors (which are samples of the gradient of the indicator function). If we apply the divergence operator to both sides of the equality, we get a standard Poisson problem given by the formula:

$$\Delta\chi = \nabla \cdot G \tag{5.2}$$

where χ is the indicator function of the model (a representation of the surface we want to extract), Δ is the Laplacian operator and G is the vector field defined by the set of points and their normal vectors. The Poisson equation is then solved to find the indicator function of the model.

The Poisson surface reconstruction algorithm is already implemented in VTK, and it is the version we used for our tests.

Surface reconstruction filter

Another algorithm that is available in VTK is the surface reconstruction filter. This filter is based on the paper of Hoppe et al. [18]. The algorithm is composed of 3 phases:

1. The first phase will generate an initial surface from the point cloud by finding the surface that minimizes the signed distance between the surface and the points of the point cloud.

2. The second phase will take the initial approximation of the surface and will optimize the mesh (reduce the number of faces) and will try to improve the fit of the surface to the point cloud. This step does not use interpolation and will only try to improve the fitting of the points themselves. This is because the point cloud is assumed to be noisy. The optimization is done by minimizing an energy function on meshes that have the same topology as the initial mesh but whose number of mesh vertices, position and connectivity can change. Minimizing the energy function should lead to a mesh with fewer vertices and a better fit to the point cloud.

3. The last phase is a piece wise smoothing of the surface that will try to keep the sharp features of the surface. The idea is to split the surface into multiple pieces and to smooth each piece separately while keeping the sharpness of the edges that link the pieces.

Unfortunately, during our testing, we found that the algorithm was too slow compared to the other tested algorithms and since time is a critical factor for our application, we decided to not test it further. We might have missed some parameters that could have improved the processing time, but since we already had other algorithms that were working, we decided to not do further tests due to time constraints.

Powercrust

The Powercrust algorithm is another filter proposed by VTK that takes a point cloud and tries to generate a surface that represents the volume. The algorithm was described by Amenta et al. [19]. The principle behind the Powercrust algorithm is to approximate the medial axis transform of the surface represented by the point cloud. The medial axis transform is a representation of the surface of the volume that is composed of the set of points that are equidistant to the surface of the volume. It can be seen as the skeleton of the model. The surface of the volume is then approximated by using an inverse transform of the medial axis transform.

Unfortunately, we didn't manage to make the algorithm work with our data and since we already had others algorithms that were working, we decided to not test the algorithm further due to time constraints.

5.4 Standalone surface extraction pipelines

As already mentioned, the second type of surface extraction algorithms are the ones that extract the surface from the volume by directly using the voxel volume as an input. Since these algorithms are more straightforward than the first type of algorithms, we will present them directly in this section and will touch upon the sub-voxel refinement step in the next section.

Some of these algorithms require defining labels on the volume. The principle of labels is to assign each voxel to a group, each with a different label. In our case, since our goal is to extract the surface of a particular feature of the volume, only a single label will be assigned to the voxels that are part of the feature that we want to extract. To define the label, our idea was to use the edge detection described in the section 5.3.1 to detect the edge of the feature that we want to extract and to assign the label to all the voxels that are inside the edge. To detect what voxels are inside the edge, a filling algorithm could be used.

In our case, since the Otsu threshold algorithm was already correctly representing all the voxels that are inside the volume, we decided to use the output of the Otsu threshold algorithm as the label of the volume. But, as already mentioned, the Otsu threshold might not give perfect results on volumes that are not as simple as the one we used for our tests. In that case, a flood fill algorithm might be needed to correctly assign the label to the volume.

5.4.1 Flying Edges

The flying edges algorithm was described by Schroeder et al. [20]. Here, the volume is divided into cubes of 8 voxels and a four pass process is applied. The first pass will iterate through the grid row by row and compute the number of intersections between the surface and the edges of the cubes in the x direction. It will mark for each row the leftmost and rightmost cubes that has an intersection with the surface. The second pass will process all cubes between the leftmost and rightmost cubes of each row and will compute the number of points and triangles that will be generated by the surface extraction depending on the number of intersections of the cube with the surface in the y and z direction. The third pass will be used to allocate the memory needed to store the points and the triangles of the surface. It also prepares the parallel execution of the final step. The final step will use a table of configurations to generate the points and the triangles of the surface.

5.4.2 Surface Nets

The Surface Nets algorithm was described by Gibson [21]. Surface Nets uses the same cube marching principle as Flying Edges in its first initial step in order to place logical cubes composed of 8 neighbouring voxels in one of the 3 following groups:

- The cube is inside the surface. This is the case when all the 8 voxels composing the cube have a value that are all in the label group.
- The cube is outside the surface. This is the case when all the 8 voxels composing the cube have a value that are all outside the label group.
- The cube is on the surface. This is the case when the 8 voxels composing the cube have a mix of values that are inside and outside the label group. A first point will be placed at the center of the cube.

This grouping step is represented in Figure 5.5.

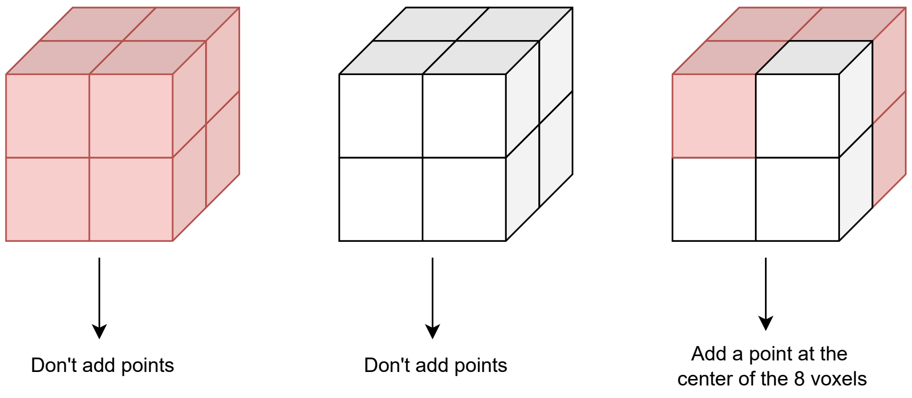


Fig. 5.5: This figure shows the process of the Surface Nets algorithm that will determine the initial point representation of the surface. In the figure, white voxels are voxels that are outside the surface and red voxels are voxels inside the surface. The last cube only represents an example of the possible cube configuration that will lead to a point on the surface. In the algorithm, any logical cube that has a mix of voxels that are inside and outside the surface will lead to a point on the surface.

This is done for all possible cubes of the volume, and neighboring points will be connected to form an initial mesh that represents the surface of the volume.

The second step of the algorithm is to refine the mesh that was generated in the first step. Each point's position will be modified in order to reduce an energy measure. Any energy measure can be used, but the one given as an example in the paper is to minimize the sum of the square of the length of the links between the points (the mesh edges). Constraints can be added to the relaxation step (which is the second step) in order to keep the new points positions consistent with the initial mesh. One constraint given in the paper is to keep each point inside its logical cube composed of 8 voxels. The process is iterative, and the number of iterations has an impact on smoothness and edge sharpness of the final mesh.

The last step of the algorithm is to triangulate the mesh that was generated in the second step. The triangulation process will check each node and its links one by one and will, a bit like the Marching Cubes algorithm, use a table to determine what triangles to generate depending on the links between the node and its neighbors.

The algorithm is already implemented in VTK, and it is the version we used for our tests. Schroeder et al. [22] goes into details about its implementation in VTK, with a focus on parallelization and compatibility with multi-label volumes.

5.4.3 Sub-Voxel refinement

The idea is to improve the precision of the surface that was extracted by the standalone surface extraction algorithms by using as much information as possible from the initial voxel volume. This step has the same goal as for the sub-voxel refinement presented in Section 5.3.2 but instead of using the points given by the Canny surface, the points of the initial mesh volume of the surface extraction algorithm will be used.

For this step, instead of using the direction of the gradient as the direction of the sampling, we decided to use the normal vectors of the surface that was extracted by the standalone surface extraction algorithms in order to reduce the deformations caused by the movement of the points of the mesh. This was only done for the first iteration of the process, since the goal of the refinement is to converge to a local maximum of the gradient.

5.5 Test set and methodology

To test the performance of our surface extraction technique, a simulation of a CT reconstruction was made on a synthetic part, of which we know all the exact dimensions. Then, measurements were made on the extracted surface and compared to the real dimensions of the volume. These measurements allow us to quantify the error made by the whole surface extraction pipeline. The tested volume is a cube with a pyramid placed on top of it and a cylinder removed at the center of the cube. The volume is shown in Figure 5.6.

The testing procedure was to measure the distance between all edge points of the cube and of the pyramid, to measure the angle between the base of the cube and one of its side, the angle between 2 adjacent sides of the cube and the angle between the base of the pyramid and one of its side.

To measure those metrics on the reconstructed surface and on the point cloud, we used a VTK function that finds the closest points/vertex to given coordinates. We used that function to find the points that are the closest to the corners of the cube and the top of the pyramid, and used them to measure the length of all edges. The set of points chosen are the orange points in Figure 5.6 and the red lines on that same figure represent the measured distances.

To compute the angles, we decided to define vectors from points of the surface and compute their dot product in order to determine the angles between them. The edges of the vectors are found on the reconstructed surface by taking the closest points to given coordinates (just like the points used to measure the edges). We decided to measure the

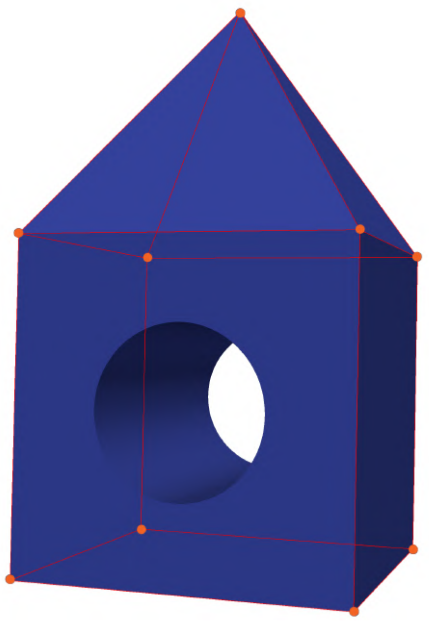


Fig. 5.6: This figure shows the model used to test the surface extraction algorithms. The model is composed of a cube with a pyramid on top of it and a cylinder removed at the center of the cube. The volume is used to test the precision of the surface extraction algorithms. The orange points are the points used to measure the distances between the edges of the cube and the pyramid, and the red lines are the distances measured.

angle between the base of the cube and one of its side, the angle between 2 adjacent sides of the cube and the angle between the base of the pyramid and one of its side. The vectors used to measure the angles are shown in Figure 5.7.

For surface extraction algorithms that are based on the point cloud computed from the application of the Canny algorithm, the measurements were made after the initial computation of the point cloud, after the sub-voxel refinement step and after the surface creation step. This allows us to quantify the improvements or the errors made at each step of the pipeline on the precision of the extracted surface.

For the standalone surface extraction algorithms, the measurements were made with and without the sub-voxel refinement step. This allows us to quantify the improvements or the errors made by the sub-voxel refinement step on the precision of the extracted surface and to compare them with the performances of the original surface extraction algorithms.

These measurements were made on both the simulated CT reconstruction and on a voxelization of the initial model. The voxelization was generated from the initial .stl model and used a voxel resolution equal to the resolution of the simulated CT reconstruction. The voxelization was made thanks to a VTK filter that generates a voxel volume from a .stl model. This was made in order to see the errors made by the surface extraction algorithms on a voxel volume that isn't affected by the noise/artifacts caused by the CT reconstruction process which represent the best case scenario for these algorithms.

On top of these precise measurements, a more visual error measure was also computed

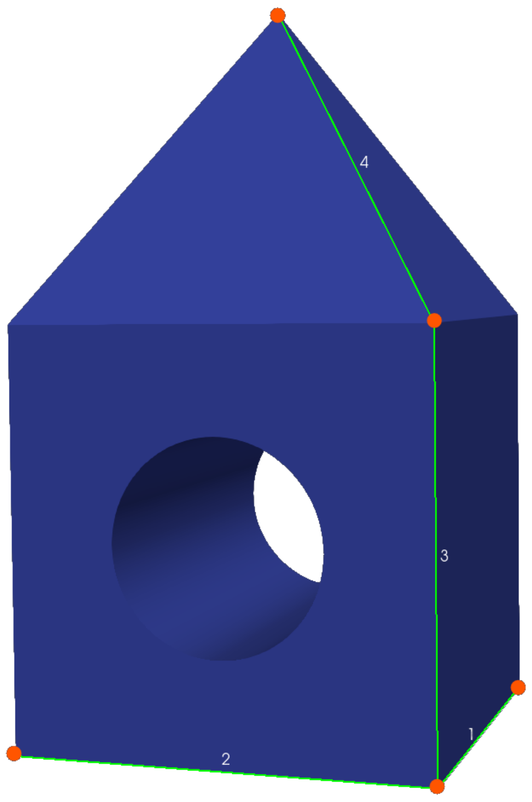


Fig. 5.7: This figure shows the model used to test the surface extraction algorithms. The green line represents the vectors used to measure the angles between the different parts of the model and each number on top of the green lines are used to identify the different vectors. The angles measured are the angle between the base of the cube and one of its side (between 1 and 3), the angle between 2 adjacent sides of the cube (between 1 and 2) and the angle between the base of the pyramid and one of its side (between 3 and 4).

for between all final surface volumes and the initial stl model. This measure is the application of the vtkDistancePolyDataFilter. This VTK filter will color the produced model depending on the point to point distance measured between each points of the two models. An example of an output of the filter is shown in Figure 5.8. From this output, we can easily see where the errors are made and how big they are. This filter also allows us to compute the mean, minimum and maximum error made on the whole surface, which is a metric that will also be used to compare the different algorithms.

5.6 Results

In this section, the results of the tests on the different surface extraction algorithms will be presented in the form of tables that will show the error made by the surface extraction algorithms on the different metrics that were measured. The results will be presented for the simulated CT reconstruction and for the voxelization of the initial model.

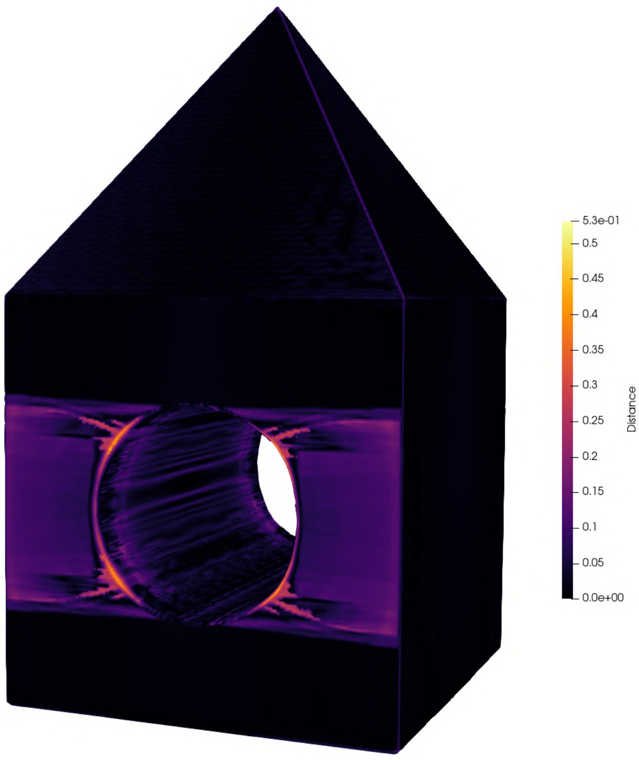


Fig. 5.8: This figure shows the output of the vtkDistancePolyDataFilter filter. The colors on the model represent the point to point distance between the output of a surface extraction algorithm and the initial .stl model. The legend on the image shows the distance that corresponds to each color. In this case, a voxel has a size of 0.2 mm.

5.6.1 Initial point cloud

For this section, we will analyze the error metrics of the initial and refined point cloud before the application of surface reconstruction algorithms. This will be useful to determine the error added or removed by both the surface reconstruction algorithms and the initial sub-voxel refinement of the point cloud.

Here, we will present the error measurement before and after the application of the sub-voxel refinement step. The results will be presented for the simulated CT reconstruction and for the voxelization of the initial model. For point clouds, we cannot apply the vtkDistancePolyDataFilter since the point cloud is not a closed surface. Thus, only the measurements made on the edges of the cube and the pyramid will be presented.

Table 5.1 shows the measurements on the point cloud before the application of the sub-voxel refinement step for both the simulated CT reconstruction and the voxelization of the initial model. As can be seen in the table, the error is the same for both. This is quite surprising, but this can be explained by the fact that the point cloud is generated by taking the center of the voxel that the Canny algorithm detects as being part of the surface and that we are only using the distance between all the corners of the cube and pyramid. Since the artifacts caused by the reconstruction are mostly present around the volume, particularly on the surface planes and that these artifacts are the only difference between the two volumes, it is not far-fetched for the Canny output of the two volumes to be the same around the cube and pyramid corners.

	Mean (mm)	Minimum (mm)	Maximum (mm)	Std dev (mm)
Reconstructed volume	0.421041	0.200001	0.599998	0.200482
Voxelized volume	0.421041	0.200001	0.599998	0.200482

Tab. 5.1: This table shows the mean, minimum, maximum and standard deviation errors made in the measurement of the edges of the cube and the pyramid on the point cloud before the application of the sub-voxel refinement step. The measurements were made on the simulated CT reconstruction and on the voxelization of the initial model.

We will now analyze the results of the measurements made on the point cloud after the application of the sub-voxel refinement step. The results are shown in table 5.2. As can be seen, for both cases, the sub-voxel refinement step managed to reduce the error made on the edges of the cube and the pyramid. For the simulated CT reconstruction, the mean error was reduced from 0.421041 to 0.262462 and for the voxelization of the initial model, the mean error was reduced from 0.421041 to 0.297767. This shows that the sub-voxel refinement step managed to improve the precision of the point cloud for the edges of the cube and the pyramid.

	Mean (mm)	Minimum (mm)	Maximum (mm)	Std dev (mm)
Reconstructed volume	0.262462	0.105112	0.534302	0.264908
Voxelized volume	0.297767	0.108801	0.451692	0.199061

Tab. 5.2: This table shows error made on the measurement of the edges of the cube and the pyramid on the point cloud after the application of the sub-voxel refinement step. The measurements were made on the simulated CT reconstruction and on the voxelization of the initial model.

Unfortunately, the error made on the edges of the cube and the pyramid are still quite high after the sub-voxel refinement step. Indeed, a voxel has a size of 0.2 mm and as such an error higher than that size means that we made an error higher than a voxel. Ideally, the error after sub-voxel refinement should be around a tenth of a voxel. As will be seen in the next sections, the error made on the surface itself will be quite low compared to the error made on the edges. This means that the problem is most likely caused either by the way the normal vectors are selected on the edges or by the way the interpolation is done in the sub-voxel refinement step around the edges.

The edge error is also higher for the voxelized volume than for the simulated CT reconstruction. We think that this is most likely caused by the way the voxelization is done. Indeed, the issue might be caused by the fact that the voxelization outputs a binary voxel volume whereas the simulated CT reconstruction outputs a volume where each voxel is defined by an unsigned short value that represents the density of the material. This difference lead to less information available when computing the gradient of the image which is used for the sub-voxel refinement step.

5.6.2 Extract surface

For the extract surface algorithm, only the results with sub-voxel refinement will be considered. The edge error and the error made on the whole surface will be used for our analysis.

Table 5.3 shows the measurements made on the surface extracted by the extract surface algorithm. As can be seen on the table, the error on the edges is reduced compared to the point cloud. This might seem surprising since the extract surface algorithm is based on the point cloud generated by the Canny algorithm. But, as can be seen in Figure 5.9, this is actually most likely caused by the rounding done by the algorithm on the edges of the cube and the pyramid. This rounding makes the edges of the cube thicker which might compensate a shrinking error made by the point cloud.

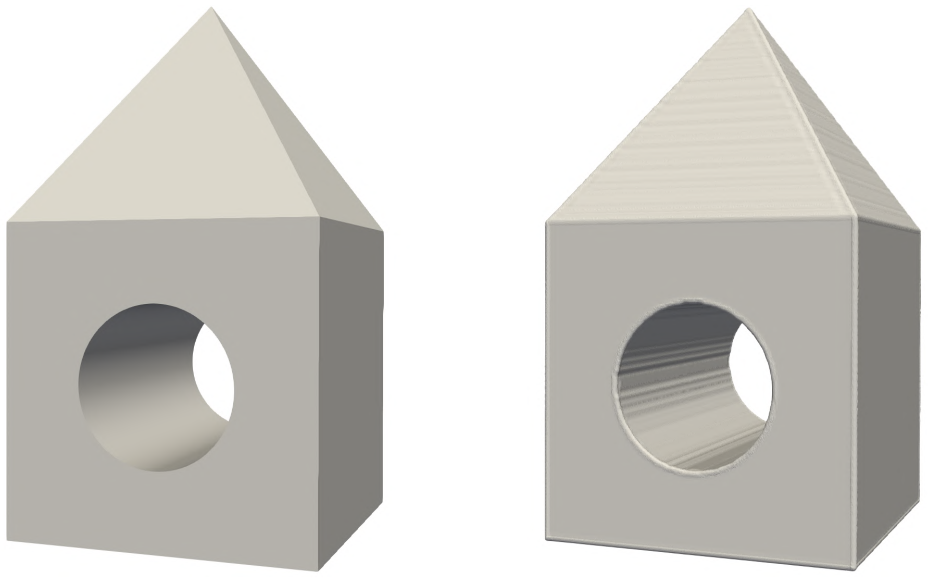


Fig. 5.9: This figure shows both the initial model on the left and the output of the Extract Surface algorithm on the voxelized representation of the volume on the right. As can be noticed, the edges of the cube and the pyramid are thicker on the output of the Extract Surface algorithm than on the initial model.

Table 5.4 shows the mean, minimum and maximum error made on the whole surface. As can be seen, that error is quite lower than the one made on the edges. As expected, the error is lower for the voxelized volume than for the simulated CT reconstruction. This is because the voxelized volume doesn't have the noise and artifacts caused by the CT reconstruction process. Figure 5.10 shows the output of the vtkDistancePolyDataFilter for both the simulated CT reconstruction and the voxelized volume. As can be seen on the figure, the artifacts caused by the CT reconstruction caused an additional error, especially on the faces of the surface.

The error made on the angle were of 0.0253332, 0.136957 and 0.148776 degrees for the simulated CT reconstruction and 0.0299201, -0.0657082 and 0.0844544 degrees for the voxelized volume.

5.6.3 Poisson surface reconstruction

For the Poisson surface extraction algorithm, the error made on the edges significantly increased as can be seen on table 5.3. We think that this is because the Poisson algorithm smooth out the sharp edges of the surface which leads to a higher error on the edges but a more appealing surface.

The mean error made on the whole surface is lower than the one of the Extract surface algorithm as can be seen on table 5.4. This is again most likely caused by the smoothing

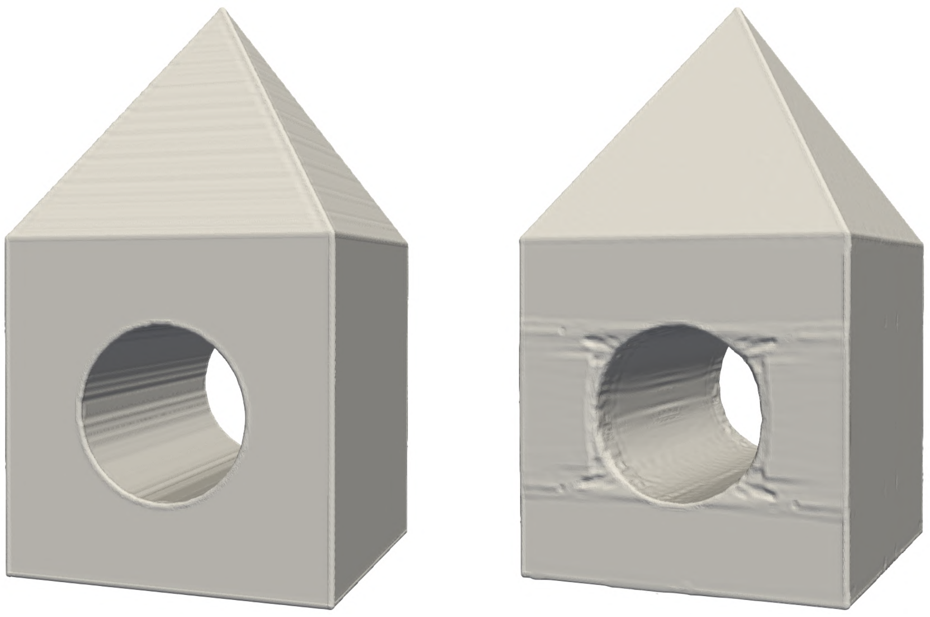


Fig. 5.10: This figure shows both the output of the Extract Surface algorithm on the voxelized volume on the left and on the simulated reconstructed volume on the right. As can be noticed, the noise and artifacts caused by the CT reconstruction process are visible on the output of the Extract Surface algorithm and not on the voxelized volume.

done by the Poisson algorithm that reduces the impact of the artifacts of the CT reconstruction. The maximum error is way bigger which can be explained by the higher error made on the edges.

The error made on the angle were of -0.348027, -0.579037 and -0.015238 degrees for the simulated CT reconstruction and 0.00921807, 0.000138 and 0.0648162 degrees for the voxelized volume. The error made on the angles is higher than the one made by the Extract surface algorithm for the simulated CT reconstruction but lower for the voxelized volume.

5.6.4 Flying Edges

A smoothing step was added to the output of the Flying Edges algorithm because the output of the Flying Edges algorithm is expected to be smoothed out to compensate its initial roughness and to have a better initial approximation of the surface normal vectors. Indeed, without the smoothing, they are noisy and the sub-voxel refinement step would output a volume with artifacts and holes as can be seen on figure 5.11.

For standalone surface extraction algorithms, we will first look at their performance without sub-voxel refinement. For the Flying Edges algorithm, the error made on the edges is smaller than the error made by the Poisson surface reconstruction algorithm with sub-voxel refinement but worse than the Extract surface algorithm as can be seen on table 5.4. As expected, the error on the voxelized volume is smaller than the one on the reconstructed volume.

For the error on the whole surface, the Flying Edges algorithm has a mean error that is worse than both the Poisson algorithm and the Extract Surface algorithm. This is not

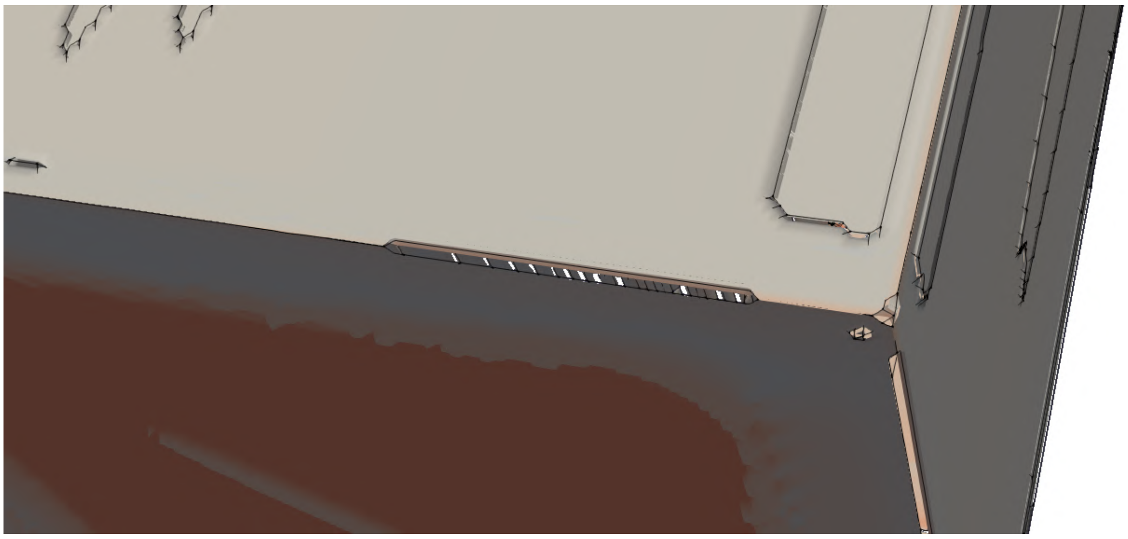


Fig. 5.11: This figure shows an example of the output of the Flying Edges algorithm without the smoothing step. The output presents holes and black artifacts.

surprising as the sub-voxel refinement step is not applied to the Flying Edges algorithm. The error is not too far off from the ones of the Poisson and Extract Surface algorithms as can be seen on table 5.4.

The error made on the angle were of 0.0556518, 0.00511808 and 0.0673448 degrees for the simulated CT reconstruction and 0.13504, -0.0534684 and -0.0378726 degrees for the voxelized volume.

We will now analyze the results with sub-voxel refinement. The mean error made on the edges is reduced from 0.405704 to 0.267444 for the reconstructed volume and from 0.376402 to 0.252407 as can be seen on table 5.3.

The mean error made on the whole surface is reduced from 0.0668432 to 0.0436207 for the reconstructed volume and from 0.0585901 to 0.0259658 as can be seen on table 5.4. This shows that the sub-voxel refinement step managed to improve the precision of the surface extracted by the Flying Edges algorithm, but the error made is still higher than a tenth of a voxel. The mean error is actually smaller than both the mean surface error of Poisson and Extract Surface algorithms.

The error made on the angle were of 0.298404, -0.0836861 and 0.102139 degrees for the simulated CT reconstruction and 0.17287, 0.0978697 and -0.00581236 degrees for the voxelized volume.

5.6.5 Surface Nets

Here, no external smoothing was enabled and instead the internal smoothing parameters of the algorithm were used. We didn't use the smoothing step added for the Flying Edges algorithm since using it would lead to issues with Surface Nets. We think that this is linked to the way the smoothing filter deals with the normal vectors of the VTK volume and how Surface Nets define them at his output, but we didn't have time to investigate further and just used the internal smoothing parameters of the algorithm.

Here again, we will begin by looking at the performance of the algorithm without sub-voxel refinement. The mean error made on the edges is smaller than the one made by the Flying Edges algorithm without sub-voxel refinement as can be seen on table 5.3. The error is still higher than the one made by the Extract Surface algorithm.

The mean error made on the whole surface is higher than the one made by the Flying Edges algorithm without sub-voxel refinement as can be seen on table 5.4.

The error made on the angle were of 0.0333227, 0.0390084 and 0.18532 degrees for the simulated CT reconstruction and -0.011602, 0 and 0.117937 degrees for the voxelized volume.

We will now analyze the results with sub-voxel refinement. The mean error made on the edges is reduced from 0.339359 to 0.241748 for the reconstructed volume and from 0.312235 to 0.228206 as can be seen on table 5.3. The error is still higher than the one made by the Extract Surface algorithm, but it is lower than the one made by the Flying Edges algorithm.

The mean error made on the whole surface is lower than the one made by the Flying Edges algorithm with sub-voxel refinement as can be seen on table 5.4. This is the case for the reconstructed volume but not for the voxelized volume. The error on the reconstructed volume is the lowest of all tested surface extraction algorithms.

The error made on the angle were of 0.0641384, 0.097213 and 0.102139 degrees for the simulated CT reconstruction and 0.0575634, -0.0651643 and -0.10234 degrees for the voxelized volume.

5.7 Discussion

In this section, we will quickly discuss the results of the tests on the different surface extraction algorithms thanks to both the error metrics tables and histograms of the error made on the whole surface in a particular case for each algorithm.

Table 5.3 shows the error each surface extraction algorithm made on the edges of the cube and the pyramid. As can be seen on the table, the error made on the edges is quite high for all the algorithms. The means error in measurement is higher or close to the size of a voxel for all the algorithms. This is not acceptable for a metrology application such as the one we are aiming for. We are not entirely sure why the error is so high on the edges, but we think it might be due to either the algorithms all having poor performance on the edges or the interpolation of the gradient on the edges not giving expected results.

The mean error made on the whole surface is lower than the one made on the edge for all the algorithms as can be seen on table 5.4. The mean error is the range of a fourth of a voxel for the Surface Nets algorithm and is the smallest mean error made for the reconstructed volume. While this is an improvement compared to the error made on the edges, it is still quite high for a metrology application. The mean error is higher for the reconstructed volume compared to the voxelized volume for all algorithms which is not surprising since the reconstructed volume has noise and artifacts than the voxelized volume does not have.

Method	Mean (mm)	Minimum (mm)	Maximum (mm)	Std dev (mm)
Extract Surface RV	0.181915	0.073752	0.270282	0.052096
Extract Surface VV	0.227572	0.087276	0.345659	0.084774
Poisson RV	0.631970	0.101470	0.980591	0.681563
Poisson VV	0.578672	0.375038	0.714992	0.153303
Flying Edges RV	0.405704	0.227589	0.511656	0.095776
Flying Edges VV	0.376402	0.202129	0.495849	0.114566
Refined FE RV	0.267444	0.052531	0.506473	0.161177
Refined FE VV	0.252407	0.154381	0.338996	0.041310
Surface Nets RV	0.339359	0.160461	0.461791	0.159002
Surface Nets VV	0.312235	0.137312	0.449883	0.201480
Refined SN RV	0.241748	0.166236	0.351523	0.039836
Refined SN VV	0.228206	0.183575	0.259425	0.011371

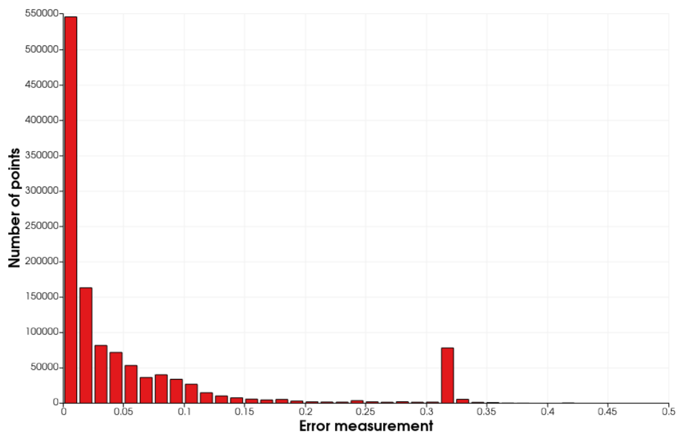
Tab. 5.3: This table shows the error in mm made on the measurement of each edge of the cube and the pyramid for the different surface extraction algorithms. The measurements were made on the simulated CT reconstruction (RV) and on the voxelization of the initial model (VV). In the method names, FE stands for Flying Edges and SN stands for Surface Nets.

Method	Mean (mm)	Minimum (mm)	Maximum (mm)	Std dev (mm)
Extract Surface RV	0.055311	0	0.746586	0.087987
Extract Surface VV	0.047794	0	0.161995	0.036438
Poisson RV	0.046987	0	0.528166	0.066606
Poisson VV	0.044322	0	0.327572	0.038916
Flying Edges RV	0.066843	0	0.573198	0.056372
Flying Edges VV	0.058590	0	0.213245	0.022501
Refined FE RV	0.043621	0	0.588009	0.065591
Refined FE VV	0.025966	0	0.216099	0.015959
Surface Nets RV	0.074068	0	0.521124	0.064041
Surface Nets VV	0.072074	0	0.214579	0.048733
Refined SN RV	0.040564	0	0.531191	0.060210
Refined SN VV	0.034029	0	0.240683	0.026414

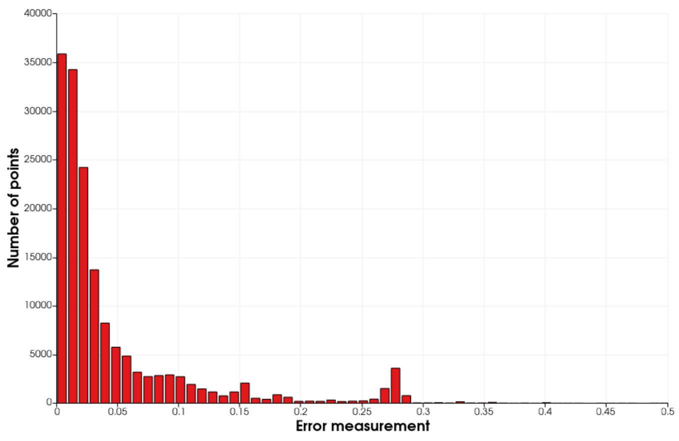
Tab. 5.4: This table shows the point to point error in mm made on the whole surface for the different surface extraction algorithms. The measurements were made on the simulated CT reconstruction (RV) and on the voxelization of the initial model (VV). In the method names, FE stands for Flying Edges and SN stands for Surface Nets.

For the histograms, we choose to only show the histogram of the error made on the whole surface for the simulated CT reconstruction for each algorithm. We also set the horizontal scale of the histograms to be the same for all the algorithms in order to ease the comparison between them. The histograms are shown in Figure 5.12.

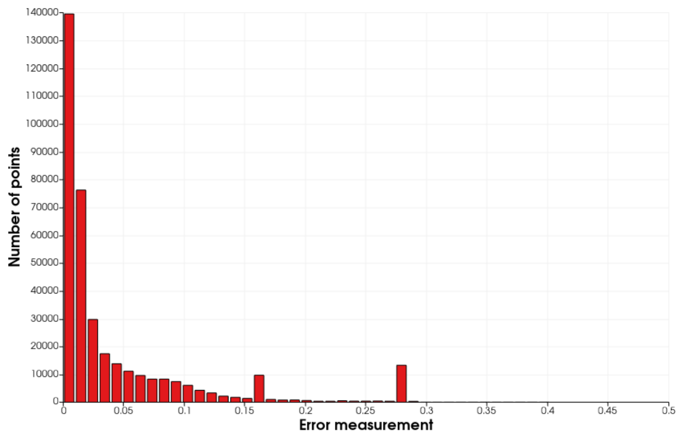
The histograms show that most of the points on the surface have an error that is lower than a voxel. We also see that all the algorithms have a second smaller peak of points with an error that is higher than a voxel. This is caused by the reconstruction artifacts located at the bottom of the cube that can be seen in Figure 5.4. Since this error is due to the incorrect filtering of the artifacts, it is normal to see it on all the histograms. We also notice that the general shape of the histograms is quite similar for all the algorithms and the difference between them is mainly their width. The Extract Surface algorithm has the



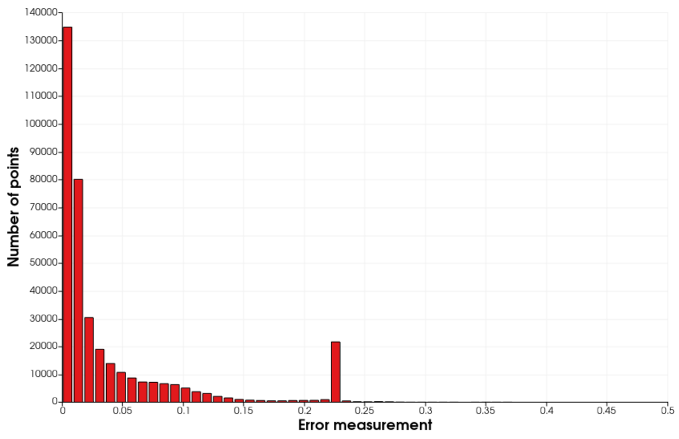
(a) Extract Surface



(b) Poisson



(c) Flying Edges



(d) Surface Nets

Fig. 5.12: This figure shows the histograms of the error measurements made on the whole surface of the reconstructed volume for the different surface extraction algorithms. The histograms were made on the simulated CT reconstruction and are shown with the same horizontal range to ease the comparison between the algorithms.

highest width which means that the error made on the whole surface is more distributed and higher than for the other algorithms. The Poisson histogram is thinner than the one of the Extract Surface algorithm but is wider then the two other histograms. Both the Flying Edges and Surface Nets histograms are similar in shape and width. The major difference between the two is the position and scale of the second peak. In Flying Edges, the second peak is split in two peaks: one that is higher than the one of the Surface Nets algorithm and one that is lower. This difference leads to the Flying Edges having a lower mean error than the Surface Nets algorithm even though the second peak of the Flying Edges algorithm is higher than the one of the Surface Nets algorithm.

5.8 Conclusion

In this chapter, we compared a multitude of surface extraction algorithms on a particular volume and from there tried to determine which algorithm would have the best performances for a metrology application. Unfortunately, the results of the tests were not as conclusive as we had hoped. Indeed, for a metrology application, accuracy is key, and the error measurements found in this chapter are not satisfactory especially for the edges of the volume as the error made on them were most of the time higher than a voxel. The global surface error, while much lower than the error made on the edges, is still higher than required.

The Extract Surface algorithm was the one algorithm that provided the best results in terms of precision of the edges for both the reconstructed volume and the voxelized volume. But as already stated in the dedicated paragraph, we think that this is more of a coincidence caused by the rounding of the edges made by the algorithm and not a real improvement of the precision of the edges.

The Surface Nets algorithm gave the best global precision on the reconstructed volume and the Flying Edges algorithm had the best global precision on the voxelized volume. But the best all around algorithm was the Surface Nets algorithm thanks to it comparable global error than the Flying Edges algorithm and its better edge precision compared to Flying Edge.

The next step of this work would be twofold: firstly, find better ways to deal with the reconstruction artifacts since the current solutions are not satisfactory and secondly, find ways to improve the precision of the reconstruction on the edges. This could be done with either another surface extraction algorithm like Dual Contouring presented by Ju et al. [23] which claims to correctly deal with sharp edges or by looking into feature recognition algorithms that could be used to improve the precision of the edges. Indeed, most manufactured pieces that would be analyzed thanks to surface extraction techniques are usually made with CAD software. Thus, pieces are usually composed of a combination of simple shapes that are extruded. A recognition algorithm could thus be used to detect these simple shapes and to improve the precision of the edges of the surface. But this is outside the scope of this second part of the work.

Chapter 6

General conclusion and perspectives

One of the goal of this thesis was to find a way to render voxel volumes that are larger than the VRAM. In Chapter 2, we presented an existing solution called GigaVoxel developed by Crassin et al. [5] that seemed like a good starting point to tackle the issue. It would split the volume into group of voxels called bricks and generate multiple representation of the volume with different levels of detail. Each representation is then stored inside a sparse voxel octree where any brick at any level of detail can easily be fetched. On the GPU, a caching pool based on page tables is used in order to allow bricks to be loaded seamlessly in and out of the VRAM depending on what was rendered. The idea is to only load visible parts of the volume into the VRAM, and to load the parts of the volume that are visualized from further away with a reduced level of detail. Since these parts are further away, the user won't be able to notice the loss of detail and since fewer details are present in those parts, they take less space inside the VRAM.

The library was improved during this thesis in a multitude of ways. The pre-processing step of voxel volumes to convert them into an octree was optimized in order to considerably reduce its runtime. The brick filtering process of the library was improved thanks to the addition of range values. This allows users to more precisely filter out the bricks that they do not want to see, which in turn improves performance thanks to the reduction of the number of bricks that are loaded into the VRAM. In addition to these changes, bug fixes and smaller improvements to the user experience were made.

In Chapter 3, the rendering modes that were added to the library were presented. These rendering modes were the one implemented by X-RIS in their Maestro software and are used to help user with NDT analysis. We thus showed that the bricks and caching system did not prevent the implementation of these rendering modes. The performance impact of these rendering modes was also analyzed and while the performance impact of them is not negligible, all are acceptable thanks to the implementation of modifiable ray step values.

In Chapter 4, we analyzed the impact of the size of the bricks on both the octree file size and the octree generation time. We also theorized on the impact that the brick size would have on the memory usage and performance. From these experiments, we concluded that the optimal brick size would be of $32 * 32 * 32$ for most use cases. We also analyzed the performance of the library on two specific datasets. One showed the usefulness of the library compared to down sampling, and a second one showed that our solution worked with satisfactory performance on a real dataset of 58 GB.

We think that the library is in a state where it can be used by the Maestro software and as such the work of the importation of the library into the Maestro software can begin. A point that could be improved in the future, is the memory usage. Indeed, since we worked with the assumption that there would always be enough system memory (RAM) available on the machine to contain the whole volume, the rendering of extremely large volumes could cause issues. It would thus be interesting to look into ways to optimize the memory usage of the library.

In Chapter 5, we investigated the topic of surface extraction techniques. These are algorithms that extract surface meshes of volumes represented by voxels. Surface meshes are useful in the field of metrology to perform measurements on the surface of the volume. We thus developed a pipeline that would allow us to extract the surface of a volume, and we tested the performances of multiple state-of-the-art surface extraction algorithms.

For this part, our work was not very conclusive but offered a starting point for future work. The main issue is that the tested methods do not offer a good enough precision for what is required in the field of metrology. Indeed, when the edges of the reconstructed surface were measured and compared to their expected values, most tested algorithms made an error bigger than a voxel. For the point to point surface error, the mean error was equal to a fourth of a voxel for the best performing algorithm which, while better than the edge error, still is higher than required. Indeed, for precise metrology application, the error should be in the range of a tenth of a voxel.

We think that future work should focus on the use of a method that better deals with sharp edges and on a better way to filter out the noise in the volume caused by the reconstruction artifacts of CT scans. An idea for possible future work would be to look into the usage of feature recognition in order to extract useful information about the volume that can be used to improve the surface extraction.

Appendix A

Configuration of the test machines

For our benchmarks, two machines were used. Their configuration will be listed here :

- Machine A:
  - CPU: Ryzen 7 7800x3D
  - GPU: RTX 4070 Super (12 GB VRAM)
  - RAM: 32 GB DDR5 6000 MHZ
  - Storage: Samsung 980 Pro
- Machine B :
  - CPU: I9 11900F
  - GPU: RTX 3090 (24 GB VRAM)
  - RAM: 128 GB DDR4 3200 MHZ
  - Storage: Samsung 980 Pro

Bibliography

[1] Klaus Engel. “CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs”. In: *2011 IEEE Symposium on Large Data Analysis and Visualization*. 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV). Providence, RI, USA: IEEE, Oct. 2011, pp. 123–124. ISBN: 978-1-4673-0155-8 978-1-4673-0156-5. DOI: 10.1109/LDAV.2011.6092330. URL: http://ieeexplore.ieee.org/document/6092330/ (visited on 09/23/2023).

[2] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. “High resolution sparse voxel DAGs”. In: *ACM Transactions on Graphics* 32.4 (July 21, 2013), pp. 1–13. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/2461912.2462024. URL: https://dl.acm.org/doi/10.1145/2461912.2462024 (visited on 09/23/2023).

[3] Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobbetti. “SSVDAGs: symmetry-aware sparse voxel DAGs”. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’16: Symposium on Interactive 3D Graphics and Games. Redmond Washington: ACM, Feb. 27, 2016, pp. 7–14. ISBN: 978-1-4503-4043-4. DOI: 10.1145/2856400.2856420. URL: https://dl.acm.org/doi/10.1145/2856400.2856420 (visited on 09/23/2023).

[4] M. Hadwiger et al. “Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (Dec. 2012), pp. 2285–2294. ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.240. URL: http://ieeexplore.ieee.org/document/6327233/ (visited on 09/23/2023).

[5] Cyril Crassin et al. “GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering”. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. I3D ’09. New York, NY, USA: Association for Computing Machinery, Feb. 27, 2009, pp. 15–22. ISBN: 978-1-60558-429-4. DOI: 10.1145/1507149.1507152. URL: https://doi.org/10.1145/1507149.1507152 (visited on 09/23/2023).

[6] Jonathan Sarton et al. “Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-of-Core Approach”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.10 (Oct. 1, 2020), pp. 3008–3021. ISSN: 1077-2626, 1941-0506, 2160-9306. DOI: 10.1109/TVCG.2019.2912752. URL: https://ieeexplore.ieee.org/document/8695851/ (visited on 09/23/2023).

[7] Lance Williams. “Pyramidal parametrics”. en. In: *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*. Detroit Michigan USA: ACM, July 1983, pp. 1–11. ISBN: 978-0-89791-109-2. DOI: 10.1145/800059.801126. URL: https://dl.acm.org/doi/10.1145/800059.801126 (visited on 04/08/2024).

[8] Aashish Chaudhary et al. "Cross-Platform Ubiquitous Volume Rendering Using Programmable Shaders in VTK for Scientific and Medical Visualization". In: *IEEE Computer Graphics and Applications* 39.1 (Jan. 1, 2019), pp. 26–43. ISSN: 0272-1716, 1558-1756. DOI: 10.1109/MCG.2018.2880818. URL: https://ieeexplore.ieee.org/document/8663650/ (visited on 12/19/2023).

[9] J. Kniss, G. Kindlmann, and C. Hansen. "Multidimensional transfer functions for interactive volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (July 2002), pp. 270–285. ISSN: 1077-2626. DOI: 10.1109/TVCG.2002.1021579. URL: http://ieeexplore.ieee.org/document/1021579/ (visited on 01/25/2024).

[10] Cyril Crassin. "GigaVoxels : un pipeline de rendu basé Voxel pour l'exploration efficace de scènes larges et détaillées". Theses. Université de Grenoble, July 2011. URL: https://theses.hal.science/tel-00650161.

[11] Timothy Lottes. *Fast Approximate Anti-Aliasing*. 2009. URL: https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf (visited on 03/24/2024).

[12] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". en. In: *ACM SIGGRAPH Computer Graphics* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: https://dl.acm.org/doi/10.1145/37402.37422 (visited on 03/24/2024).

[13] J.A. Yagüe-Fabra et al. "A 3D edge detection technique for surface extraction in computed tomography for dimensional metrology applications". In: *CIRP Annals* 62.1 (2013), pp. 531–534. ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2013.03.016. URL: https://www.sciencedirect.com/science/article/pii/S0007850613000176.

[14] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

[15] Nobuyuki Otsu. "A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66. DOI: 10.1109/TSMC.1979.4310076.

[16] Brian Curless and Marc Levoy. "A volumetric method for building complex models from range images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. ISBN: 0897917464. DOI: 10.1145/237170.237269. URL: https://doi.org/10.1145/237170.237269.

[17] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. "Poisson surface reconstruction". In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 4. 2006.

[18] Hugues Hoppe et al. "Surface reconstruction from unorganized points". In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '92. New York, NY, USA: Association for Computing Machinery, 1992, pp. 71–78. ISBN: 0897914791. DOI: 10.1145/133994.134011. URL: https://doi.org/10.1145/133994.134011.

[19] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. “The power crust”. en. In: *Proceedings of the sixth ACM symposium on Solid modeling and applications*. Ann Arbor Michigan USA: ACM, May 2001, pp. 249–266. ISBN: 978-1-58113-366-0. DOI: 10.1145/376957.376986. URL: https://dl.acm.org/doi/10.1145/376957.376986 (visited on 05/26/2024).

[20] William Schroeder, Rob Maynard, and Berk Geveci. “Flying edges: A high-performance scalable isocontouring algorithm”. In: *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*. 2015, pp. 33–40. DOI: 10.1109/LDAV.2015.7348069.

[21] Sarah F. F. Gibson. “Constrained elastic surface nets: Generating smooth surfaces from binary segmented data”. In: *Medical Image Computing and Computer-Assisted Intervention — MICCAI’98*. Ed. by William M. Wells, Alan Colchester, and Scott Delp. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 888–898. ISBN: 978-3-540-49563-5.

[22] Will Schroeder et al. *A High-Performance SurfaceNets Discrete Isocontouring Algorithm*. 2024. arXiv: 2401.14906 [cs.GR].

[23] Tao Ju et al. “Dual contouring of hermite data”. In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 339–346. ISSN: 0730-0301. DOI: 10.1145/566654.566586. URL: https://doi.org/10.1145/566654.566586.