TU WIEN Informatics

Homomorphic-Encrypted Volume Rendering

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Sebastian Mazza, BSc
Matrikelnummer 00825828

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr. techn. Assoc. Prof. Ivan Viola
Mitwirkung: Assoc. Prof. Daniel Patel, PhD

Wien, 7. Mai 2021

_____   _____
Sebastian Mazza   Ivan Viola

TU Wien Informatics

Homomorphic-Encrypted Volume Rendering

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Sebastian Mazza, BSc

Registration Number 00825828

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr. techn. Assoc. Prof. Ivan Viola
Assistance: Assoc. Prof. Daniel Patel, PhD

Vienna, 7th May, 2021

_____   _____
           Sebastian Mazza   Ivan Viola

Erklärung zur Verfassung der Arbeit

Sebastian Mazza, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Mai 2021

Sebastian Mazza

Plagiarism Disclaimer

The core of this thesis has been present at the *IEEE VIS 2020* conference and is published in the *IEEE Transactions on Visualization and Computer Graphics* journal. The manuscripts of the journal paper [47] has been written as part of the master thesis course. Therefore, this thesis and the journal paper [47] share the same content (text, figures, tables and algorithms). The initial problem statement that led to this work is from Daniel Patel, since he raised the question if it was possible to perform a volume rendering directly on encrypted data that keeps the volume data private. The author of this thesis took up the key idea from Daniel Patel and developed it to the presented Homomorphic-Encrypted Volume Rendering approach. The ideas, program code implementations, and the manuscript for the journal mainly originate from the author of this thesis, but were developed in collaboration with the supervisors of this thesis, namely Ivan Viola and Daniel Patel. They have supported the progress of this thesis, contributed to uncountable discussion, encouraged creative and unconventional approaches, and assisted the writing process by providing valuable feedback and reviews. Hence, the supervisors Ivan Viola and Daniel Patel are also the co-authors of the journal paper.

Associated Journal Publication

Danksagung

Ich bedanke mich bei Christoph Heinzl für das Zurverfügungstellen seiner industriellen CT-Scans [25] und bei P. Li, S. Wang, T. Li, J. Lu, Y. HuangFu und D. Wang, die ihre PET- und CT-Scans für die Lungenkrebsdiagnose [42] im Cancer Imaging Archive (TCIA) [5] zur Verfügung gestellt haben. Wir haben diesen Volumendatensatz verwendet, um unseren Ansatz zum Rendern von verschlüsselten Volumesdatensätzen unter realen Bedingungen evaluieren zu können.

Ich danke Ivan Viola und Daniel Patel dafür, dass sie die besten Betreuer sind, die sich ein Student wünschen kann. Sie gaben mir die notwendige Zeit, um Probleme und mögliche Lösungen zu untersuchen. Sie haben mir immer wieder geholfen, zurück auf die richtige Spur zu finden, wenn ich mich in den falschen oder irrelevanten Details verrannt hatte. Ohne ihr ständiges Feedback und ohne den endlosen Diskussionen mit ihnen wäre diese Arbeit nicht möglich gewesen.

Ivan möchte ich weiters dafür danken, dass er mich dazu ermutigt hat, ein wissenschaftliches Manuskipt für die IEEE VIS-Konferenz 2020 zu schreiben. Er war großartig darin, mich während des gesamten Einreichungs-, Überprüfungs- und Präsentationsprozesses immer wieder zu motivieren. Ohne die vielen, oft auch nächtlichen, Evaluierungen durch Daniel und Ivan wäre ich nicht in der Lage gewesen, sämtliche Abgabefristen einzuhalten. Das ganze VIS-Paper-Projekt war zwar ein erheblicher zusätzlicher Aufwand, aber auch eine großartige Erfahrung und ich bin Ivan dankbar, dass er mir als Master-Student diese Erfahrung ermöglicht hat.

Auch meinem Freund Carl möchte ich für seine Unterstützung, nicht nur während des Schreibens dieser Arbeit, sondern auch während meiner gesamten Zeit an der Universität danken. Besondere Anerkennung verdient er für die vielen Stunden, in denen er meine diversen Manuskripte Korrektur gelesen hat. Er war für mich auch der Englischlehrer, den ich leider in der Schule nie hatte. Ich möchte ihm auch dafür danken, dass ich monatelang bei ihm wohnen durfte. Auch die unzähligen inspirierenden, interdisziplinären, mehr oder weniger wissenschaftlichen Diskussionen möchte ich nicht unerwähnt lassen.

Zu guter Letzt danke ich meiner Familie, insbesondere meiner Tante Franzi, meinem Onkel Herbert und meiner Mutter Ilse, für ihre unendliche Unterstützung und Motivation während der gesamten Zeit. Ein ganz besonderer Dank geht an Edith, die mich auf vielfältige Art und Weise unterstützt hat.

Gewidmet meinem Vater.

Acknowledgements

Dedicated to my father.

Kurzfassung

Rechenintensive Aufgaben werden oft in Rechenzentren verarbeitet und auch die Echzeit-Visualisierung folgt diesem Trend. Allerdings erfordern einige Rendering-Aufgaben ein Höchstmaß an Sicherheit, damit niemand außer dem Eigentümer die vertraulichen Informationen lesen oder sehen kann. Wir präsentieren in dieser Arbeit einen Ansatz zum direkten Rendern von Volumen-Datensätzen, bei dem die für das Rendern notwendigen Berechnungen direkt auf den verschlüsselten Volumendaten mithilfe des homomorphen Paillier-Verschlüsselungsalgorithmus durchgeführt werden. Dieser Ansatz stellt sicher, dass die Volumendaten und die gerenderten Bilder für den Server, der das Rendering ausführt, nicht interpretierbar sind. Unsere Volume-Rendering-Pipeline führt neuartige Ansätze zur Komposition, Interpolation und Transparenz-Modulation von verschlüsselten Dichte-Informationen ein, und zeigt eine Möglichkeit zum einfachen Design von Transferfunktionen auf. Dabei gewährleisten all diese neuen Routinen ein Höchstmaß an Datenschutz. Wir präsentieren eine Analyse für den Leistungs- als auch Speicher-Overhead, der mit unserem den Datenschutz gewährleistenden Rendering-Ansatz verbunden ist. Unser Ansatz ist offen und sicher durch sein algorithmisches Design und versucht nicht Sicherheit durch das Verbergen von Implementierungsdetails vorzutäuschen. Die Eigentümer der Daten müssen lediglich ihren privaten Schlüssel geheim halten, um die Vertraulichkeit ihrer Volumendaten und der gerenderten Bilder zu gewährleisten. Unseres Wissens nach repräsentiert diese Arbeit den ersten Ansatz, der den Schutz der Daten beim entfernten Rendern von Volumendaten garantiert, auch wenn keiner der involvierten Server als vertrauenswürdig betrachtet wird. Selbst wenn der Server kompromittiert ist, ist es für Fremde nicht möglich, Zugang zu sensiblen Daten zu erlangen. Darüber hinaus haben wir eine Big-Integer (Zahlen die länger als native Maschinenwörter sind) Bibliothek für die Vulkan-Grafikpipeline entwickelt. Diese ermöglicht das Rendern mit sicher verschlüsselten Daten auf der GPU. Die entwickelte Bibliothek unterstützt die Berechnungen gängiger mathematischer Operationen wie Addition, Subtraktion, Multiplikation und Division. Weiters werden spezielle Operationen für die asymmetrische Kryptographie, wie zum Beispiel die modulare Potenzierung mit Montgomery-Reduktion, unterstützt. Außerdem zeigen wir noch ein Testframework, das wir zum automatisierten Testen von Big-Integer Berechnungen auf der GPU entwickelt haben.

Abstract

Computationally demanding tasks are typically calculated in dedicated data centers, and real-time visualizations also follow this trend. Some rendering tasks, however, require the highest level of confidentiality so that no other party, besides the owner, can read or see the sensitive data. Here we present a direct volume rendering approach that performs volume rendering directly on encrypted volume data by using the homomorphic Paillier encryption algorithm. This approach ensures that the volume data and rendered image are uninterpretable to the rendering server. Our volume rendering pipeline introduces novel approaches for encrypted-data compositing, interpolation, and opacity modulation, as well as simple transfer function design, where each of these routines maintains the highest level of privacy. We present performance and memory overhead analysis that is associated with our privacy-preserving scheme. Our approach is open and secure by design, as opposed to secure through obscurity. Owners of the data only have to keep their secure key confidential to guarantee the privacy of their volume data and the rendered images. Our work is, to our knowledge, the first privacy-preserving remote volume-rendering approach that does not require that any server involved be trustworthy; even in cases when the server is compromised, no sensitive data will be leaked to a foreign party. Furthermore, we developed a big-integer (multiple-precision, or multiple word integer) library for Vulkan graphics pipeline. It facilitates the rendering of securely encrypted data on the GPU. It supports the calculation of common mathematical operations like addition, subtraction, multiplication, division. Moreover, it supports specialized operations for asymmetric cryptography like modular exponentiation with Montgomery reduction. We also introduce a testing framework for Vulkan that allows the automated testing of big-integer computations on the GPU.

Contents

Introduction

Volume rendering is extensively used in domains where the underlying data is considered highly confidential. One example includes the field of medicine, where CT, MRI, or PET data are used for diagnostic or treatment-planning purposes. Another such example is hydrocarbon and mineral exploration in energy industries for inspecting the subsurface using seismic scans.

For volume rendering, privacy can currently only be achieved by storing and processing the datasets locally. Volume rendering requires computers with large memory and powerful processing power. Such hardware must be frequently maintained and upgraded. Therefore, for many organizations, it would be advantageous to outsource the rendering to cloud services. As cloud services remove the need to be in close proximity to the rendering hardware, users can now also view volume rendering on thin clients that do not have the required memory or processing power, such as tablets and smart phones. However, hospitals must protect sensitive personal data and energy companies must protect their valuable data assets. Thus, it is essential that their data is not visible to the cloud services, as these either cannot be trusted, or their security might be compromised. Therefore, we want to make it possible to perform direct volume rendering on untrusted hardware while preserving the same level of privacy for the datasets as the privacy achieved with a classical local rendering approach.

The basic concept of our privacy-preserving approach is shown in Figure 1.1. First, the data is acquired and immediately encrypted by, for example, a machine that is directly connected to a medical scanner. Then the encrypted volume is uploaded to the *honest-but-curious*[1][59] public server. This is done only once per volume. When the clients that hold the secure key request rendering, the server performs ray-casting directly on the encrypted volume data. This computation results in an image containing encrypted

---

[1]The server will perform the algorithm it is requested to compute honestly (correctly); however, there is the potential for possible access by the curious eyes of administrators or hackers.
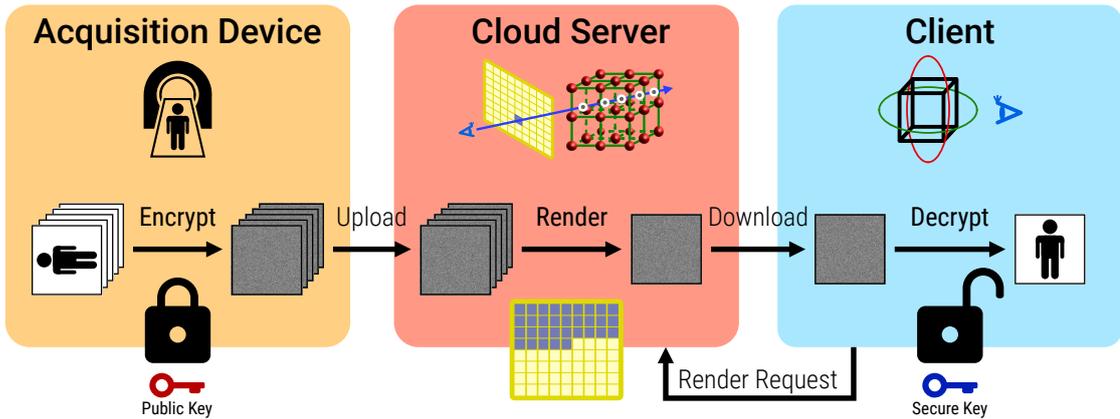
Figure 1.1: Our approach consists of a computer that produces, encrypts, and sends volume data to a server, which then renders the data and sends the result to a client. The client decrypts and visualizes the result.

values, which is then sent to the client. When the client receives the requested image, it is decrypted and displayed to the user. As the server that computed the rendered image will only see encrypted data, our approach maintains privacy.

Our design is constrained by three requirements. The *first requirement* is that the privacy of the user data is protected by the design of the algorithm and does not depend on hiding implementation details, keeping any part of the system secret, or any other obscure technique that cannot be secure, at least not in the long run (Kerckhoffs's principle [32]). Such obscure techniques only make it difficult to know the actual security of the system. Therefore, the security of our volume rendering approach solely depends on the security of a well-established cryptographic algorithm, continuously being scrutinized in cryptographic research. We have chosen the well-established Paillier cryptographic algorithm, which is partly homomorphic [58]. The key property of homomorphic encryption (HE) is that arithmetic operations on encrypted data are dual to arithmetic operations on *plaintext* (original, unencrypted) data. This enables an algorithm to perform a correct 3D volume rendering image synthesis directly on the encrypted data, without being able to ever access the plaintext data. As a consequence, the result of the rendering on the server is an encrypted image.

We are currently not able to show interactive frame rates with the proof-of-concept implementation of our approach. Neither in Java on the CPU nor with Vulkan on the GPU. However, one future research goal should attempt to make a remote rendering system fast enough to achieve this. This leads to our *second requirement*, which is to only use techniques that will not prevent the system from scaling the performance with the computational power available on the server and will not prohibit interactive frame rates. The *third requirement* is to support thin clients without much memory and computational power. As a result, we consider the client to be a low powered device, which is connected to a mobile or another medium-bandwidth network, while we assume that the server is a

powerful machine (e.g., with multiple professional GPUs) or even a compute cluster.

By using encryption schemes like AES [13], it is currently possible to store volume datasets securely in the cloud. However, for rendering images from the datasets, the entire volume needs to be downloaded and decrypted first, and then rendered on the client. A privacy-preserving remote volume rendering can also make the cloud more attractive as a storage space for volume data, because with our proposed technique, it is no longer necessary to download the whole dataset before images can be synthesized from it.

1.1 Motivation for Big-Integer Computation on the GPU

Asymmetric or homomorphic encryption schemes like RSA [66] or Paillier [58] requires integer arithmetic for numbers that are longer than the common machine word sizes of todays processors. While usual processors supports 32 bit or 64 bit arithmetics, asymmetric cryptography requires calculations with integers that are thousands of bits long. Therefore, multiple machine words (e.g. an unsigned integer in C++) must be used together in order to store an integer that exceeds the maximal native machine word size. We will call an integer that is longer than the native machine word size and constructed from multiple machine words: *Big-Integer*.

In order to prove our ideas about homomorphic encrypted volume rendering we implemented a prototype in Java. We have chosen Java for that task, because it is easy to write, the debugging is simple and most important, it supports integer arithmetics with nearly infinite length out of the box through its `java.math.BigInteger` class [57]. Furthermore, Java `java.math.BigInteger` class supports many advanced algorithms that are important in the context of asymmetric cryptography. Like quickly finding prime numbers that are thousands of bits long, efficient modular exponentiation ($a^x \mod n$), or finding the multiplicitive inverse of an integer in a residues class ($a^{-1} \mod n$ - see *modInverse* at the Section 6.3.6).

However, the homomorphic encrypted volume rendering approach we will introduce in this work (Chapter 4), is based on the ray casting algorithm (Section 2.1.1), which is highly parallelizable. Therefore, the homomorphic encrypted volume rendering should scale well for many processing units. Consequently, GPUs should be fitting perfectly because they have lots of processing units. Yet, a GPU is not a CPU with hundreds of cores. While a CPU core can execute program code independently from any other CPU core a GPU can only execute a very limited set of different instructions at a given time. The processing units of GPU (streaming processors, threads) are grouped together in work groups (compute units, blocks in CUDA). Such a work group can only perform a single instruction at a given time, but every streaming processor can apply the instruction to its own set of input variables. Therefore, a GPU consists of so called single instruction, multiple data (SIMD) processors that can apply the same operation to hundreds of input operants simultaneously. Without going into too much detail we can state that GPUs are highly price efficient for workloads that apply the same operation

to many input data in parallel. Nonetheless, the programming for GPUs is often more complicated than for CPUs and not every procedure can be implemented efficiently on GPUs. Algorithms, where the flow control depends on the individual input data of a thread, is a potential problem on a GPU. While modern GPUs can handle situations where half of the streaming processors of a work group take the `if` path and the other half the `else` path of a program flow, this is still at least an inefficient situation. Since the work group first executed the instruction from the `if` path on all stream processors and only disables the data storage for the stream processors that should go through the `else` path of the procedure. After all instructions in the `if` path are executed the workgroup enables the output writing for the stream processors that should go through the `else` path and disables the writing for the stream processors that should take the `if` path and then executes the instructions of the `else` program block. The performance drop in that case is at most two. The things are getting worth if the two distinct program flow blocks contain further branching. Imagine a while loop where 63 streaming processors are finished after the first iteration but one streaming processor requires 100 iterations. This situation basically means that 63 streaming processors resources are wasted for the time it takes to execute the body of the loop 99 times. Therefore, not all algorithms that can be efficiently executed in parellel on multiple CPU cores, are also efficient for the execution on a GPU.

Workstation, server, and even some high end smartphone processors support native machine words of up to 64 bits, common smartphones only support 32 bits. While modern high performance GPUs generally support 64 bit arithmethics, 32 bit integer arithmetics seams to be about 4 times faster (see AIDA64 GPGPU benchmarks from Serve the Home [21], tested cards: NVIDIA RTX 2060, RTX 2080, RTX 3090, Quadro 6000, Quadro 8000, AMD RX 5700 XT, Radeon Vega Frontier Edition, RX 6800 and more). From the AIDA64 GPGPU benchmarks made by Serve the Home [21] another important observation can be made while comparing the integer and float operations per second. While on Nvidia cards, the count of 24 bit and 32 bit integer operations per second is between the half and the same as the single precision floating-point operations per second, on AMD cards the 24 bit integer operations are as fast as single precision floating-point operations, but the 32 bit integer operations are 5 times slower than the single precision floating operations on AMD cards.

There are already quite a few works on the subject of big-integer on GPU (Section 2.3.2), but only very few freely available libraries and almost all of them are for CUDA. One exception is the C++ library from Blake Warner [78] which is designed for OpenCL. While the advantage of CUDA is clearly the relative low entry barriers for developers, the disadvantage is that it is only supported on Nvidias GPUs and a very limited set of platforms. On the other hand Vulkan is an open standard which is supported on many more devices and platforms (e.g. Android, iOS, Linux, Mac OS, Windows), while it is also a modern and performant GPU API. Therefore, we started to develop an open source big-integer library for Vulkan.

4

1.1.1 Developed Open Source Big-Integer Library for Vulkan

The developed library is available at `https://github.com/vzz3/HEVolRender`. It includes the required host code and the GLSL code for all common mathematical operations and also some specialized methods for asymmetric cryptography (see Section 6.3). Furthermore, it includes a unit testing framework (see Section 6.5.2) for GPU code and an example application that implements the whole homomorphic encrypted X-ray rendering pipeline.

1.2 Contributions

The novel contributions of this work are summarized in the following points:

- Best to our knowledge we have implemented the first working prototype for a privacy preserving volume rendering that is secure by design and features X-ray compositing scheme.

- We show a flexible approach for (tri)linear interpolation of homomorphic encrypted values.

- We show the first practical approach for a transfer-function that can transfer homomorphic encrypted density values of a voxel to RGB colors.

- Best to our knowledge we implemented the first big-integer library for Vulkan.

In Chapter 2 we provide the foundation of this work as well as an overview of related and previous works in the field of direct volume rendering (Section 2.1), privacy preserving computation (Section 2.2) and big-integer arithmetics (Section 2.3). Chapter 3 contains a recap about homomorphic encryption in general and Paillier in detail. Then we introduce our encrypted rendering pipeline in Chapter 4. Furthermore, this chapter contains an explanation on how to use an encrypted floating-point representation for trilinear interpolation. In Chapter 5 we will first discuss the difficulties at designing a transfer function approach for values that are encrypted with a probabilistic encryption scheme. Followed by the introduction of our simplified transfer function approach. Chapter 6 contains the implementation details of our big-integer GPU library and the encrypted X-ray rendering with Vulkan. The achieved results are shown in Chapter 7. It contains images with example rendering results from real world datasets as well as measured runtimes for different rendering techniques, processors, and public key sizes. In Chapter 8 we provide a discussion about the performance, further improvements, and security aspects of the introduced encrypted volume rendering. This thesis will be closed by the conclusion in Chapter 9.

Related Work

The homomorphic encrypted volume rendering approach is build upon different scientific disciplines. Therefore, this chapter is structured in three sections. The first section explains the required foundations about conventional direct volume rendering. The Section 2.2 provides a summary about all previous attempts to create a remote volume rendering approach that is able to preserve the privacy of the rendered data. Furthermore, it contains an outline of other areas of computer science where homomorphic encryption was used to protect the security of sensitive data. Since the calculations with encrypted values, which are used for the presented volume rendering approach, requires integers longer then native machine words. The Section 2.3 contains a overview about big-integer computations. We discuss some interesting CPU libraries as well as previous works about big-integer computations on the GPU.

2.1 Volumes Rendering

A volumetric dataset can be seen as a three-dimensional space-filling grid. A "point" in such a three-dimensional grid is called *voxel* and typically contains a scalar value. For this work we will assume that each voxel of a volumetric dataset is a scalar value (e.g. the object density at the position of the voxel). Volumetric datasets are for example retrieved from Computed Tomography (CT) or Magnetic Resonance Imaging (MRI).

For a volumetric dataset, surface rendering methods like polygon rasterization can not be used to visualize the data. At least not directly, because it is possible to extract surfaces out of volumetric datasets and render these surfaces with surface rendering methods. However, this gives more limited information to the viewer compared to rendering the volume data directly

The typical user goals in the context of direct volume rendering are to gain insight in 3D data and finding structures of special interest (e.g., a doctor that wants to identify a
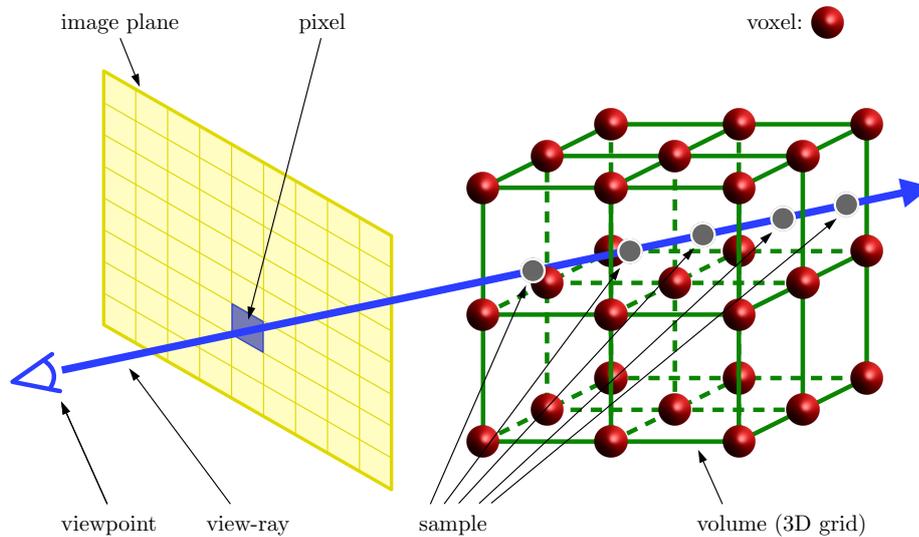
Figure 2.1: Illustration of ray casting

pathology in the MR scan of a patient).

A straightforward way to investigate volume data is slicing. This is done by displaying a plane that cuts through the volume, where all voxel values that lie on the plane are mapped to an image.

2.1.1 Ray Casting

More sophisticated approach was developed by Drebin et al. [12] and Levoy [41]. Both papers describe a way to project a volume dataset to an image plane. However, Drebin et al. transforms and resamples the volume in order to show the volume from different directions, while Levoy uses ray casting. The idea behind ray casting can be described as shooting a ray for every pixel through a volume (see Figure 2.1). At discrete and equidistant steps along the ray, the data of the volume is sampled. The most simple way to do that is to align the image along two of the three volume axes (e.g., x and y) and sample along the third axis (e.g., z) of the volume. The result is an orthogonal projection of the volume that is aligned with one of the axes of the volume. Figure 2.1 illustrates the sampling of a viewing ray with an arbitrary origin and direction.

Perspective projections can provide a better 3D impression for the user, because the perspective distortion encodes depth information in a form that is natural to the human brain. In order to help the users to investigate and understand the volumetric dataset, the users can change the parameters of the perspective projection like the viewpoint of

the projection and the field of view. By allowing the users to move the eye points of the projection interactively around the volume, they are able to observe the shape of objects inside the volume from different directions. The Appendix Section A.1 contains an algorithm that shows how to calculate the origin and direction of a view for a perspective projection.

2.1.2 Sample Compositing

In order to obtain a color for a pixel from the samples of a viewing ray, the sample values need to be combined. A simple approach for such a combination is to calculate the average of all samples along a viewing ray and display it as a grey value. This is also called X-ray compositing. Figure 2.2 shows an example of an *X-ray* rendering.
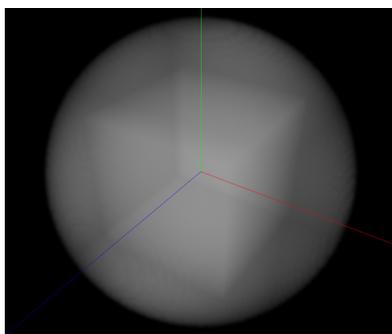
Figure 2.2: Direct volume rendering result of a volume that contains a cube with high values inside a sphere with lower values using X-ray compositing.

The α-compositing approach uses Porter/Duffs's *over* or *under* operator [63] for blending semi transparent samples together. For back-to-front ray traversal the *over* operator is used while for front-to-back rendering the *under* operator is required (see Drebin et al. 1988 [12]). α-compositing can provide a more expressive visualization of a volume and the objects it contains than an *X-ray* rendering. For α-compositing colors (RGB) and transparencies (A) are used as sample values of a ray. Therefore, the scalar density value of a voxel needs to be mapped to a vector that contains three color channels (read, green, blue) and the transparency. We call this vector RGBA. This density to RGBA mapping can be achieved by using a transfer function. (See Section 2.1.3 for more information about transfer functions.) During a ray traversal, the RGBA values at the sample positions are accumulated to the color of a pixel in the final image. In order to be able to see correct occlusion order in the volume, the accumulation needs to be implemented as an alpha blending. For an alpha blending every color value that should be added needs an information that defines how opaque the sample is. The more opaque a sample is, the more it affects the color of the final pixel. The required opacity information is provided by the alpha channel (A) of the RGBA vector.

The Equation 2.1 shows the alpha blending required for front-to-back ray traversal (the

sampling starts at the position of the observer).

$$
\begin{aligned}
C_{\mathrm{ray}} &= C_{\mathrm{ray}-1} + (1 - \alpha_{\mathrm{ray}-1})\,\alpha_{\mathrm{sample}}\,C_{\mathrm{sample}} \\
\alpha_{\mathrm{ray}} &= \alpha_{\mathrm{ray}-1} + (1 - \alpha_{\mathrm{ray}-1})\,\alpha_{\mathrm{sample}}
\end{aligned}
\tag{2.1}
$$

C_{sample} denotes the color of the current sample and α_{sample} is the alpha value of the current sample. $C_{\mathrm{ray}-1}$ and $\alpha_{\mathrm{ray}-1}$ contains the already accumulated color and alpha values, both should be initialized with 0 at the beginning of a ray traversal. As follows from the Equation 2.1, not only the already accumulated color C_{ray} needs to be maintained for the ray traversal, but also the already accumulated alpha value α_{ray}. For a monochromatic rendering (e.g. gray scale) the C variables (C_{sample}, C_{ray}, $C_{\mathrm{ray}-1}$) can be scalar values, however, for colorful results the C variables need to be vectors (e.g. RGB). In the case that the C variables are vectors the multiplications and additions need to be performed element-wise.
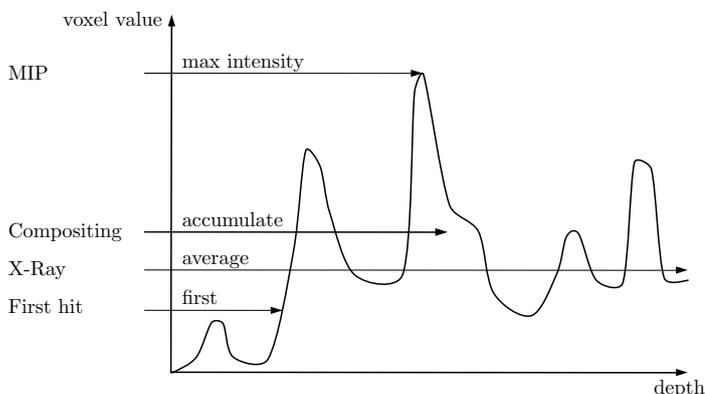
Figure 2.3: Types of voxel combinations. (Adapted from [20])

Another example for a voxel combination scheme is *first hit* (Tuy 1984) which returns only the value of the first voxel having a density above a certain threshold. Figure 2.3 shows an illustrative comparison of the mentioned voxel combination schemes.

2.1.3 Transfer Function

To provide a more versatile rendering, it is useful to be able to remap the volume values to user-defined values so that the user can make for example certain value-intervals of a CT scan corresponding to certain tissue types, which stand out in the resulting rendering. Such a mapping is called a *transfer function* (denoted as *classification functions* in Drebin et al. [12]). For non encrypted data a transfer function can be seen as a lookup table. If the volume dataset contains only a limited number of different voxel values, this lookup table can be implemented as an array that has all possible voxel values as indices and the lookup value as the data in the array. For a volume dataset that encodes each voxel as a 10 bit integer, the required array has indices from 0 to $2^{10} - 1 = 1023$. The values of

such an array usually contains three or four 8 bit values. If just a color is required three 8 bit values are used for the color channels red, green, blue. This is for example sufficient for an X-ray rendering with colors. An α-*compositing* on the other hand would require another 8 bit for the alpha value that encodes the transparency for the mapped voxel.

Another approach to implement a transfer function could be to store only some supporting points of the transfer function and perform a linear interpolation of the four color values between the two neighboring supporting points of the sample value that should be mapped. Figure 2.4 shows a user interface which allows a user to create a transfer function.
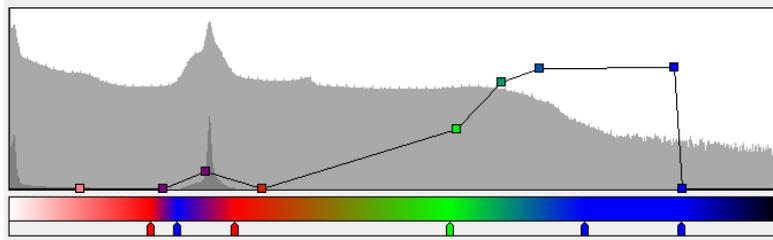
Figure 2.4: Transfer function widget; density values increasing from left to right, the different colors at the bottom visualize the colors to which the density values will be mapped. The line above it shows the level of transparencies; in the background a histogram of all voxel values is shown.

A transfer function allows a volume render engine to assign different color values to different density value ranges. Together with the possibility to assign different levels of transparency to different density values it is possible to show only structures that are important for the current user task as opaque or semi-opaque objects with user-defined colors. Furthermore, not so important surrounding objects can be shown with low opacity which can provide a context for the important objects. This allows a volume rendering system to create ghosted views. More about ghosted views and smart visibility techniques can be found in the work of Viola et al. [76].

2.1.4 Sampling

In this section we want to explain two different methods for calculating the value of a sample on a ray. The first and simpler method is the *nearest neighbor* sampling which uses the value of the voxel that has the shortest distance to the sample position as the value of the sample. The second method *Trilinear interpolation* takes the 8 voxels around the sample position into account and weights each voxel value by the distance between the voxel center and the sample position. This method results in smoother final images than those produced by nearest neighbor sampling.

The fundamental concept of trilinear interpolation is the same as for linear interpolation, but instead of interpolating between two values in one dimension, trilinear interpolation interpolates between eight values in three dimensions. Figure 2.5 shows a cube with the values $v_{000}, v_{001}, v_{010}, v_{011}, v_{100}, v_{101}, v_{110}$ and v_{111} at the corners. The gray point

indicates the sample position x, y, z for which the value v is interpolated. First, the distance in each dimension x_d, y_d, z_d between the left - front - bottom corner x_0, y_0, z_0 (value: v_{000}) and the sample position x, y, z need to be calculated and normalized for the size of the cube. The length of the cube in each direction is the difference between the left - front - bottom corner x_0, y_0, z_0 and the right - back - top corner x_1, y_1, z_1.

$$
\begin{aligned}
x_d &= (x - x_0)/(x_1 - x_0) \\
y_d &= (y - y_0)/(y_1 - y_0) \\
z_d &= (z - z_0)/(z_1 - z_0)
\end{aligned}
\tag{2.2}
$$

The values v_{00}, v_{01}, v_{10} and v_{11} are the result of four linear interpolations along the x-axis (left to right).

$$
\begin{aligned}
v_{00} &= v_{000} \cdot (1 - x_d) + v_{100} \cdot x_d \\
v_{01} &= v_{001} \cdot (1 - x_d) + v_{101} \cdot x_d \\
v_{10} &= v_{010} \cdot (1 - x_d) + v_{110} \cdot x_d \\
v_{11} &= v_{011} \cdot (1 - x_d) + v_{111} \cdot x_d
\end{aligned}
\tag{2.3}
$$

Linear interpolation along the y-axis (front to back) results in v_0 and v_1.

$$
\begin{aligned}
v_0 &= v_{00} \cdot (1 - y_d) + v_{10} \cdot y_d \\
v_1 &= v_{01} \cdot (1 - y_d) + v_{11} \cdot y_d
\end{aligned}
\tag{2.4}
$$

The final value v is the result of the linear interpolation between v_0 and v_1 along the z-axis (bottom to top).

$$
v = v_0 \cdot (1 - z_d) + v_1 \cdot z_d
\tag{2.5}
$$

2.1.5 Shading - Illumination

An important factor of the human three dimensional perception of objects is the amount of light that is reflected on the surface of an object. A particular location on a surface looks brighter, if more light is reflected from the surface into the eye of the observer. If less light reaches the eye of the observer from a particular location, this location will appear darker. Therefore, the rendering of three dimensional scenes that should look natural, need to simulate the distribution of light in the scene. Furthermore, visualizing the illumination of a surface accentuated the surface which helps a user to perceive the curvature of an object. The curvature of an object can provide a lot of information about the object.

In reality not only the light sources that are visible from a particular point s on a surface influences the color of this point s but also every surface that is visible from this point s,
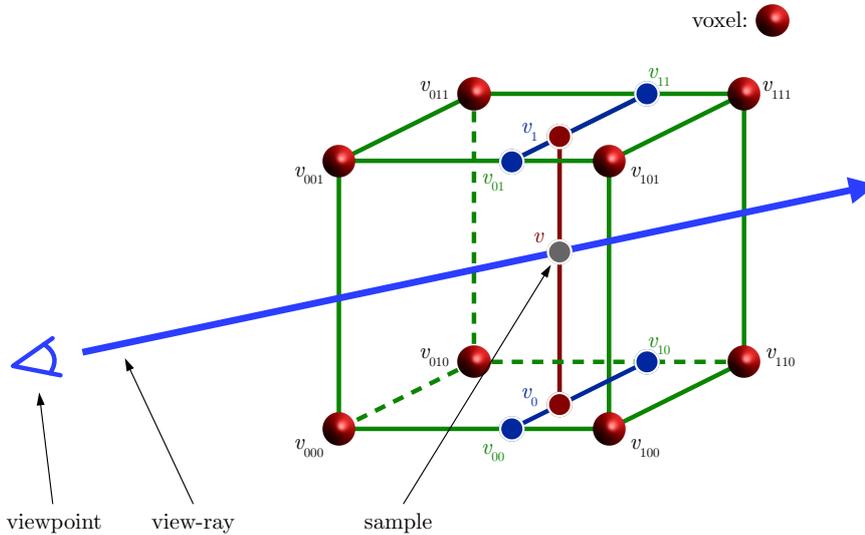
Figure 2.5: Illustration of trilinear interpolation which shows the eight corner points (voxels) on a cube surrounding the interpolation point c (sample). (Adapted from [79])

because it can reflect light to the point s. Furthermore, the point s also reflects light to all surfaces that are visible from the point s. Therefore, simulating all reflections that are happening in a scene would require an infinite amount of calculations. While there are rendering techniques (e.g. path tracing [30]) that simulate the light transport in a scene so precisely that the result is no longer distinguishable from a photograph, these techniques are out of the scope of this work. We will shortly discuss the basics of *Phong shading* [61], which was suggested to use for volume rendering by Levoy [41] and Drebin et al. [12] as an efficient approximation of the illumination of a point. It is computationally simple enough to be calculated in realtime. While it is only a simplified model of the real illumination, it provides far more accurate results than volume renderings that do not take any illumination into account (see the comparison in Figure 2.6).

The basic illumination model we will present contains three parts. Each of them simulates another lightning effect. The first part is the *ambient-reflection therm*. It approximates the light that comes from diffuse reflection on other objects and is reflected again into the eye of the observer. The ambient-light can be seen as a background light source that uniformly illuminates all objects in a scene. The ambient-light intensity is defined by the I_a. The amount of ambient reflection of an object is the result of the product:

$$k_a I_a \tag{2.6}$$

The ambient-reflection coefficient k_a defines the amount of ambient light that is reflected

by a particular object. In the case of monochromatic rendering it will contain a scalar value near 0.0 for dark objects and a value near 1.0 for bright objects. The ambient-reflection therm is independent from the position of the light source and the observers position. Therefore, the ambient-reflection term alone would produce a flat shading (see Figure 2.6a).

The *diffuse reflection therm* of the model approximates the amount of light that is scattered uniformly with equal intensity in all directions. Therefore, it is independent from the position of the observer, but it depends on the angle ϕ between the surface normal **n** and the direction to the light source **l** (see Figure 2.7). The amount of light that is reflected is proportional to $\cos(\phi)$. Therefore, the diffuse reflection can be modeled by:

$$k_d I_d (\mathbf{n} \cdot \mathbf{l}) \tag{2.7}$$

k_d denotes the diffuse-reflection coefficient and defines how reflective the surface is. The

(a) flat   (b) illuminated
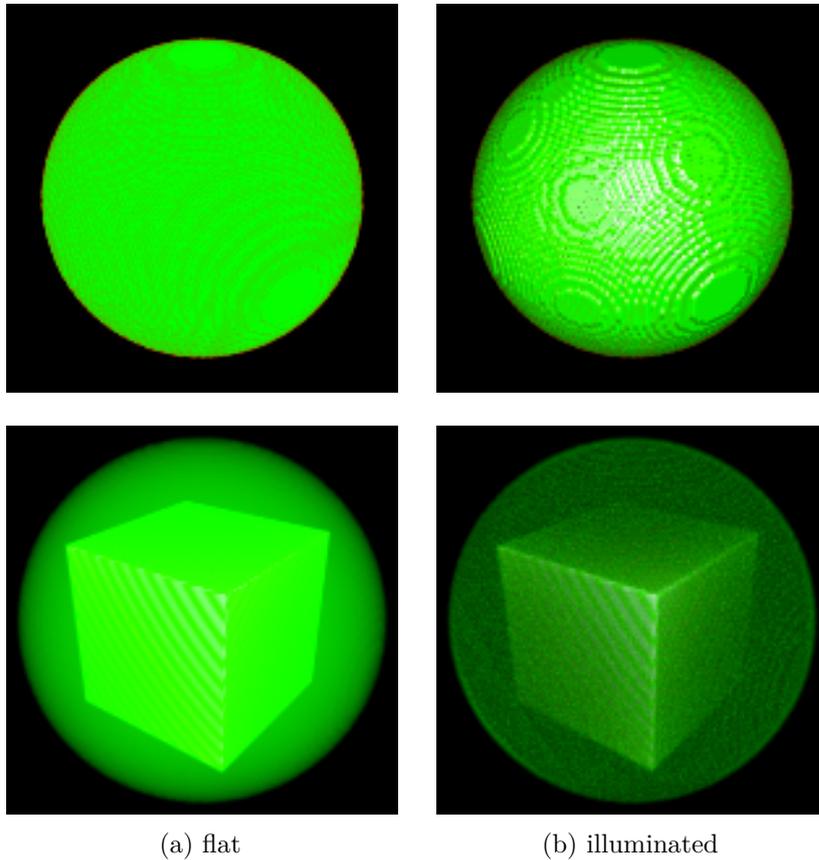
Figure 2.6: Comparison of a α-*compositing* rendering with and without illumination from a volume dataset that contains a cube in a sphere. The difference between the first row and the second row is achieved by different transfer functions.

diffuse-light intensity I_d describes the brightness of the light source.

The *specular-reflection term* simulates an effect of glossy surfaces that look like a bright spot. This effect appears in areas where the light from the light source is reflected almost perfectly into the eye of the observer. Therefore, the specular reflection depends on the angle between the direction **r** that perfectly reflects the ray of light and the direction **v** which points towards the viewer (see Figure 2.7). The reflection direction **r** can be calculated by:

$$\mathbf{r} = (2\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l} \tag{2.8}$$

The specular reflection term can be stated as:

$$k_s I_s (\mathbf{n} \cdot \mathbf{r})^{n_s} \tag{2.9}$$

k_s denote the specular-reflection coefficient (e.g. 1.0) and the diffuse-light intensity I_s is like I_d the brightness of the light source. The specular-reflection exponent n_s is the shininess constant for the material, which is larger (100 or more) for surfaces that are shiny and small (down to 1) for duller surfaces. When this constant is large, the specular highlight is small.

Suming up the ambient-, diffuse- and specular-reflection terms provides the final intensity I:

$$I = \underbrace{k_a I_a}_{\text{ambient}} + \underbrace{k_d I_d (\mathbf{n} \cdot \mathbf{l})}_{\text{diffuse}} + \underbrace{k_s I_s (\mathbf{n} \cdot \mathbf{r})^{n_s}}_{\text{specular}} \tag{2.10}$$
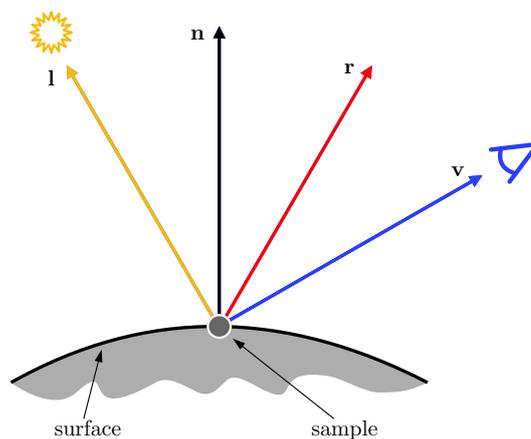
Figure 2.7: Surface normal vector **n**, view vector **v**, light vector **l** and reflection vector **r** (Adapted from [24]).

For simplicity we can assume that $I_a = I_l$ is the intensity of one light source. The coefficients k_a, k_d and k_s can contain the material color which can be an RGB value from the transfer function. If RGB values are used the coefficients are vectors which need to be multiplied component-wise with the intensity values.

In case of volume rendering the surface normal $\mathbf{n}$ which is required for the illumination calculation can be approximated by the gradient $\nabla f = (\partial f / \partial x, \partial f / \partial y, \partial f / \partial z)$ of a voxel value, because this corresponds to the normal vector of an iso-surface that could be extracted from the volume dataset.

The central difference in x-, y- and z-direction could be used as approximation for the gradient [41]:

$$\nabla f(x, y, z) \approx \frac{1}{2} \begin{pmatrix} f(x+1, y, z) - f(x-1, y, z) \\ f(x, y+1, z) - f(x, y-1, z) \\ f(x, y, z+1) - f(x, y, z-1) \end{pmatrix} \tag{2.11}$$

2.1.6 Volume Rendering on GPUs

By using modern graphic processors for direct volume rendering, the rendering can be done in real time, which allows an interactive exploration of volume datasets. There are different methods that allow direct volume rendering on the GPUs. The common idea is to draw some kind of proxy geometry that gets rasterized by the rendering pipeline of the GPU. The actual volume rendering logic is then mainly implemented in the fragment shader to which the voxel data is provide from a 3D texture. The different approaches differ in the type of proxy geometry.

Cullip and Neumann [8] described GPU accelerated slicing, which uses proxy geometry in form of planes (see *Implementing volume rendering using 3D texture slicing* in *OpenGL – Build high performance graphics* [54]). These parallel planes intersect the volume dataset in discrete intervals and can be either normal to the viewing direction or axis aligned with the volume dataset. During the rendering of the planes the fragment shader re-samples the voxel of the volume by using the filtering capabilities of GPUs for textures. After the sampling the voxel values are transferred to colors and mapped to the plane. If the planes are drawn in correct order the blending stage of the graphics pipeline can be used to blend the transferred samples together.

The rendering technique of our homomorphic encrypted volume rendering prototypes is based on the approach from Krüger and Westermann [39]. It is a GPU based ray casting that uses the bounding box of the volume as proxy geometry. The approach makes also use of 3D texture hardware for the volume storage. For each frame the bounding box is drawn three times, whereby two different types of rendering passes are used. The first type calculates and saves the volume surface positions, where each viewing ray enters and exits the volume. The second type performs the actual ray casting.

At the first pass the back face culling for back faces is enabled, which means only the faces that are directed to the viewer, are drawn. During the rendering the fragment

positions of the surface are stored as RGB color into the first target texture. The red channel contains the x-axis, the green channel the y-axis and the blue channel the z-axis. The second render pass stores the back faces of the bounding box to another target texture. Therefore, during the second render pass the back faces culling is enabled for the front faces, so, only the back face will be rasterized. Every pixel of the front face and the back face texture belongs to a screen pixel with the same pixel coordinate. The position that is stored as RGB color (3D vector) in each pixel of the front face texture represents the position where the viewing ray of that screen pixel enters the volume. The pixels of the back face texture contain the positions where the rays exits the volume. The difference between the 3D vectors stored at the same pixel position in the back face and the front face texture then represent the viewing ray direction for each screen pixel. In the third render pass the fragment shader uses the volume entry position and the direction of the ray to sample the voxel of the dataset along the viewing ray.

On current GPUs, ray casting can also be implemented with a single render pass. The basic idea is to write a shader that first calculates the ray direction by subtracting the camera position from the vertex position of the bounding box proxy geometry. The vertex position is used as ray origin. After calculating the ray origin and direction, the sampling along the viewing ray can be performed in the same render pass. Further details and an example implementation for OpenGL can be found in Section *Implementing volume rendering using single-pass GPU ray casting* of *OpenGL – Build high performance graphics* [54].

2.2 Privacy-Preserving Rendering

We have only found two works that address the topic of privacy-preserving rendering of volumetric data. The most similar work to ours is that of Mohanty et al. [50]. They present a cryptosystem for privacy-preserving volume rendering in the cloud. Unlike our approach, they achieve correct alpha compositing. However, to attain this goal, they end up with a solution that cannot be considered secure, that has a fixed transfer function, and that requires that the volume is sent from one server to another server for each rendered frame.

Their approach requires two servers for rendering: a Public Cloud Server and a Private Cloud Server. The first step of their rendering approach is to apply a color and opacity to each voxel before encrypting the volume. This means that the transfer function is pre-calculated and cannot be changed by a user without performing a time-consuming reencryption and uploading of the volume. In the next step, the encrypted data is uploaded to the Public Cloud Server, which stores the volume data. When the Public Cloud Server receives an authorized rendering request from a client, the server calculates all sample positions for the requested ray casting and interpolates the encrypted color and opacity values for each sample position. All interpolated sample values then need to be individually sent to the Private Cloud Server, which decrypts the opacity value of each sample in order to perform the alpha blending along the viewing rays. For

alpha compositing, the opacity values of samples represent object structures in the volume; therefore, anyone who can gain access to the Private Cloud Server, such as an administrator or a hacker, will be able to observe these structures in the volume dataset. If an unauthorized person has access to this server, the whole approach collapses. For the task of encrypting and decrypting parts of the volume data on the servers, their approach requires a central Key Management Authority (KMA). While this brings the advantage that an organization can centrally control which users have access to a specific volume, it enlarges the attack surface of their system considerably, because the KMA has all keys required for decrypting all volume data. Therefore, the confidentiality of the KMA is constitutional for the privacy of all datasets, no matter who they belong to.

Another weakness of their approach is the required network bandwidth between the Public and the Private Cloud Server because all sample values of a ray casting frame need to be transferred from the Public and the Private Cloud Server (more than 1GB). With our approach, the privacy of the volume data and rendered image depends only on a single secure key. Also, our approach should scale linearly with the computing power of the hardware it is running on.

Chou and Yang [4] present a volume rendering approach that attempts to make it difficult for an unintended observer to make sense of the volume dataset that resides on a server. This is done by, on the client's side, subdividing the original data into equally sized blocks. The blocks are rearranged in a random order and then sent to the server as a volume. The server then performs volume rendering on each block and sends the result back to the client, which will reorder the individual block renderings and composite them to create a correct rendering. To obfuscate the data further, on the client's side, the data values in each block are changed using one out of three possible monotonic operations: flipping, scaling, and translating. Monotonic operations are used as they are invertible and associative under the volume rendering integration. Therefore, doing the inverse operators on the resulting rendering gives the same result as doing them on the data values before performing the rendering. This algorithm cannot be considered safe, and the authors acknowledge this as they state that the goal is only to not trivially reveal the volume to unauthorized viewers. A possible attack would be to consider the gradient magnitude of the obfuscated volume. This should reveal the block borders. The gradient magnitude can further be used inside each block to reveal structures in the data that can be used for aligning the blocks correctly.

To attain our goal of developing an approach that is open and secure by design, we use the Paillier cryptosystem developed by Paillier in 1999 [58]. This cryptosystem is an asymmetric encryption scheme, where the secure key contains two large prime numbers p and q, and the public key contains the product N (modulus) of p and q. The cryptosystem supports a homomorphic addition of two encrypted values and a homomorphic multiplication between an encrypted value and a plaintext value. Paillier can securely encrypt many values (e.g., 512^3 voxels of a volume) from a small number space (e.g., 2^{10} possible density values), because it is *probabilistic*, which means that during the encryption, the *obfuscation* can map a single plaintext value randomly to a

large number of possible encrypted values. This makes a simple "probing" for finding out the number correspondence impossible. Further details about Paillier's cryptosystem such as the encryption and decryption algorithm is provided in Section 3.1. We are limited to the arithmetic operations supported by Paillier for creating a volume rendering that captures as much structure as possible from the data. This forces us to think unconventionally and creatively when designing the volume renderer.

For homomorphic image processing, the work by Ziad et al. [83] makes use of the additive homomorphic property of Paillier's cryptosystem. They demonstrate that they are able to implement many image processing filters using the limited operations allowed with Paillier. They implement filters for negation, brightness adjustment, low pass filtering, Sobel filter, sharpening, erosion, dilation and equalization. While most of these filters are computed entirely on the server side, erosion, dilation, and equalization require the client for parts of the computation. There are various works that make use of such a *trusted client protocol* approach to overcome the limitation of a PHE scheme and enable operations such as addition, multiplication, and comparisons on the encrypted data [11, 72, 10]. A *trusted client* knows the secure key and can, therefore, perform any computation on the data or *convert / re-encrypt* it from one encryption scheme to another (e.g., from an additive to a multiplicative *homomorphic encryption*). These client-side computations introduce latency because the data needs to be transferred back and forth between the server and the client. Furthermore, the client needs to have enough computational power to avoid becoming the bottleneck of the system. To mitigate this problem, automated code conversions can be used that minimize the required client side *re-encryptions* [10, 11]. While a *trusted client* approach could theoretically solve many of the problems we face with our untrusted server-only approach, it is not practical for volume rendering. The most demanding problems of volume rendering, such as transferring a voxel value and advanced compositing (alpha blending, maximum intensity projection, ...), need to be done per voxel. Hence, every voxel that could contribute to the image synthesis (all voxels of a volume for many rendering cases) needs to be transferred to the *trusted client* and processed there for every rendered frame. The encryption and decryption on the client side are more expensive than the operations required for a classical sample compositing due to the size of encrypted values (e.g., 1000 bit per voxel). If an amount of data in the range of the volume itself needs to be transferred from the server to the client, where the data would need to be encrypted and decrypted, it is pointless to perform any calculations on the server, because the client then has more work to do than in a classical volume rendering on the client. Moreover, it does not save any network traffic as compared to a simple download, decrypt, and process use case. Therefore, we argue that *trusted client* approaches are not suitable for our work. Furthermore, a *trusted client* approach will not work with thin clients, which contradicts our third requirement (see Chapter 1). Our second requirement is also contradicted because, in real-world use cases, the network bandwidth between a client like a tablet computer and a cloud server will not have enough bandwidth (e.g., more than 1 Gbit/s) to support interactive frame rates.

2.3 Big-Integer Arithmetic on GPUs

Arithmetic operations for homomorphic encryption schemes like RSA [66] or Paillier [58] require integers that are hundreds or thousands of bits long. Since usual processors support only arithmetics with 32 bit or 64 bit long integers, multiple machine words need to be used together in order to store longer integers. We will call such a multiple words long integer a *big-integer*. Obviously the storage of *big-integers* is not enough and algorithms that can perform arithmetic operations like addition $(a + b)$, subtraction $(a - b)$, multiplication $(a \cdot b)$, and division $(\frac{a}{b})$ on *big-integers* are needed. However, usually children learn algorithms that can solve this problem already in basic school. If we would only store the digits from 0 to 9 in a single machine word, we could use the classical school algorithms for addition, subtraction, and multiplication. For the guess work part of the division the procedure needs to be a bit more structured than we did at school, but the basic idea of the school division algorithm also works on a computer. However, storing only 10 digits in machine word that is 32 bits long and for that reason can store $2^{32} = 4,0294,967,296$ different values that can be interpreted as 2^{32} different digits, is a waste of storage and computational resources. An efficient approach is to change the base (or radix) of the number system from 10 to 2^{32}, if the machine word that is used for storage is 32 bit long. For other machine word sizes the radix needs to be adopted accordingly to the count of different values that can be represented by that word type. So, a *big-integer* is represented as an array of machine words, where each word contains a "digit" of the number that should be represented by the *big-integer*. Such a "digit" is between 0 and the radix minus one (e.g.: $2^{32} - 1$). This number representation is called *fixed radix number system (FRNS)*. Other number representation systems such as the *residue number systems (RNS)* has been developed and proven to be useful for certain applications. However, we do not discuss them because all our work is done around FRNS. Instead we want to refer the interested reader to the book Computer Arithmetic Algorithms by Koren [38].

For a human it is hard to perform the classical arithmetic algorithm with bases other then 10, because we are so used to writing numbers with the base 10. However, the algorithm works with any base greater or equal to 2 just as the work for the base 10. Knuth [35] also stated that fact and provides a formal definition of the classical algorithms for addition, subtraction, multiplication, and division. This concept of using multiple machine words to represent a number that exceeds the maximal length of a native machine word, that we call *big-integers* ([57, 78]), is also called Long Number ([62, 70]), Long Integers [23, 9], Multiple-Precision Integer ([51, 82, 16, 35, 14]), or Arbitrary-Precision [40] in literature.

2.3.1 Fixed Size and Arbitrary Size Integer Libraries

Two different types of big-integer libraries need to be distinguished. Libraries where the integers can grow nearly infinitely and libraries that deal with integers that can be longer than integers supported by the hardware but still limited to a predefined length. Libraries that can store nearly infinite large numbers, only limited by the

available main memory, need to use as much bytes of storage as it is needed to store the result of a particular operation. Consequently, such libraries require a dynamic memory allocation for the storage of a single big-integer. We call this type of library as *arbitrary big-integer*. Example for such libraries are Colin Plumb's C library [62], Java `java.math.BigInteger` class [57], the The GNU Multiple Precision Arithmetic Library [16] and also the CUDA library from Langer [40] from 2015, which is best to our knowledge the first *arbitrary big-integer* library for a GPU.

Another concept of big-integer libraries is referred as *fixed big-integer*. These are libraries like the C++ libraries TTMath [70], Blake Warners library for OpenCL [78] or GPUMP for CUDA [82] that can process integers that are larger by magnitudes than the 32 or 64 bit integers that are usually supported by processors in hardware. However, the length of an integer is still limited to a predefined size, which is defined during compile time. That means that arithmetic operations can produce an overflow. On the other hand, this has the advantage that the memory required for storing any big-integer is known during compile time, which allows the compiler to reserve the memory of an integer on the stack and costly dynamic memory allocations are not required, which probably gives modern compiler more opportunity for code optimization. Nonetheless, compared to *arbitrary big-integer* library the maximal length of an integer that can be practically used with *fixed big-integer* library is quite limited because the integers need to fit into the stack of a particulary operating system, at least for a CPU implementation on a common desktop platform. The frequently asked questions of TTMath [70] state that integers larger than $32,000$ bit could be a problem depending on the operating system. *Fixed big-integer* are more efficient on GPUs, not only because dynamic memory allocation are expensive but also because *arbitrary big-integers* require in principle more branching which has a negative impact on the runtime.

2.3.2 Previous Big-Integer Libraries for GPUs

Emmart [14] distiguished two different types of papers that cover big-integer arithmetic on GPUs. In the first group he listed papers that are about approaches for speeding up some very specialized routines required by specific asymmetric cryptography schemes such as *elliptic curve cryptography* [37, 49], *RSA* [66], *Diffie-Hellman key exchange* [48]. The second group Emmart defined are the papers that discuss general purpose big-integer arithmetic libraries on GPUs. One of the first paper that utilizes the GPU for big-integer calculations was done by Moss, Page, and Smart [53]. They presented a fixed big-integer 1024 bit modular exponentiation for RSA. This work is not only remarkable because it is one of the the first approaches for big-integer compute on GPUs but also because it is one of the very few works that did not use CUDA. Moss et al. used OpenGL and GLSL for their implementation. They used textures as storage for big-integer where one pixel contains one word. A single fragment shader execution performs an operation on one single word (pixel). By drawing all pixels of a big-integer an arithmetic operation can be applied to entire big-integers. In order to prevent the problem of carries across "pixels", they made use of residue number systems (RNS) where an addition, subtraction, and

multiplication does not create carries (See chapter 11 "The Residue Number System" in *Computer Arithmetic Algorithms* from Koren [38] for details.). The work of Harrison and Waldron [22] is another example for such specialized big-integer on GPU computation. They also try to speed up the RSA decryption for 1024 bit long public keys, but they did not only use an RNS number representation but also an FRNS and compared them. The first general purpose big-integer GPU library is GPUMP from Zhao and Chu in 2010 [82]. The library is implemented with CUDA and supports many operations, namely comparison, addition, modular addition, subtraction, modular subtraction, multiplication, modular multiplication, division, and modular exponentiation. GPUMP supports this operations for fixed big-integers at a length of 256 bits, 512 bits, 1024 bits, and 2048 bits.

There are also multiple-precision floating-point arithmetic libraries for GPUs. Lu et al. [43] present among other things performance comparison for different storage concepts for multiple-precision floating-point numbers on the GPU. They also used CUDA for their performance testings, just like the CUDA Multi-Precision library (CUMP) presented by Nakayama and Takahashi in 2011 [55].

The Doctoral Dissertation from Emmart [14] provides a very comprehensive and critical literature review that also contains a performance analysis of the different approaches, for which the author estimates the runtime results of different papers on other graphics hardware in order to make the results comparable.

2.3.3   Parallelizing Strategies

There are two basically different concepts how to make use of the parallel execution capabilities of a processor for big-integer libraries. The first concept is to assign only a part of an arithmetic operation to one thread and, therefore, let many threads work on a single arithmetic operation such as an addition or a multiplication (*one operation on many threads*). While this approach has the advantage of being able to speed up a single arithmetic operation, it has the disadvantage of having a lot of overhead and, therefore, not being efficient. At most operations the carry propagation between the threads is responsible for this inefficiency.

The other basic concept of performing big-integer arithmetics in parallel is to just submit one arithmetic operation to one thread. So a single thread executes all processor instructions required for performing a single big-integer operation like an addition or a multiplication (*one operation on one thread*). This approach does have way less overhead than the *one operation on many threads* approach. However, it requires x independent problems in order to be able to make use of x available processing units. We can summarize that the *one operation on one thread* approach has a high throughput (good) but also a high latency (bad). On the other hand the *one operation on many threads* approach has a low throughput (bad) but also a low latency (good). Therefore, a *one operation on many threads* is advantageous if only a view independent operations on big-integers need to be performed, while a *one operation on one thread* approach is more optimal if many independent big-integer operations need to be calculated. Harrison and

Waldron [22] did come to this conclusion and Emmart [14] stated the importance of this observation. Furthermore, Dick stated in his bachelor's thesis [9] that he was not able to make significant speedups with a *one operation on many threads* approach in CUDA and, therefore, he switched to an *one operation on one thread* approach.

Background on Homomorphic Encryption

Homomorphic encryption schemes allow computations on encrypted data such that the decrypted results are equal to the result of a mathematical or logical operation applied to the corresponding plaintext data. Therefore, calculations on encrypted data can be performed without decrypting the data first. This property makes it possible to outsource not only the storage of encrypted data, but also the computation on sensitive data to untrusted third parties (e.g., the cloud). The computation on cloud servers has two major advantages. If only the result and not the whole dataset needs to be transferred to the client, a lower network bandwidth is required. Furthermore, the client can be a thin client like a tablet without much computing and storage resources because the computational expensive rendering is done on the server. Homomorphic encryption schemes are classified into three categories:

- *partially homomorphic encryption (PHE)*: are homomorphic with regard to only one type of operation (addition or multiplication)
- *somewhat homomorphic encryption (SHE)*: Can perform more general calculations than PHE, but only a limited number of them.
- *fully homomorphic encryption (FHE)*: Can perform any computation on encrypted data.

Rivest et al. [65] invented the idea of homomorphic encryption in 1978. They showed the demand of a secure homomorphic encryption scheme that supports a large set of operations. However, it took more than 30 years until the first proposal of such a FHE was made by C. Gentry in 2009 [17]. While the first FHE schemes were just concepts, due to an ongoing development and optimization process, they are currently at least efficient enough for functional implementations.

However, FHE schemes are still not practical for real applications, because the storage and computational costs are too high [27, 46, 1]. Therefore, in our approach, we propose to take advantage of the Paillier PHE scheme, whose homomorphic properties that are relevant for our encrypted volume rendering will be introduced next.

3.1 The Paillier Cryptosystem

Paillier's cryptosystem [58] is an additive PHE scheme. The important property of this scheme is that a multiplication of two encrypted numbers is equivalent to an addition in the plaintext domain. This means that it is possible to calculate the sum of plaintext numbers that have been encrypted by multiplying the encrypted numbers. This relation is stated in Equation 3.1. For Equation 3.1 and Equation 3.2, we adopt the notation by Ziad et al. [83]. The $\oplus$ symbol is used for the operation on encrypted numbers that is equivalent to an addition on plaintext numbers. The encrypted version of the value m is denoted as $[\![m]\!]$ and $\text{Dec}([\![m]\!]) = m$ means decrypting $[\![m]\!]$ to m by the decryption function of Paillier's cryptosystem (see: Algorithm 3.3). Paillier works on finite fields and thus performs a modulo $(\text{mod } N^2)$ operation after each multiplication ensures to stay in the field and does not change the decrypted result because the corresponding plaintext numbers need to be less than the modulus N for a correct decryption anyway. However, the modulo operation makes further calculations more efficient because it prevents unnecessary large numbers (n multiplications will blow up the number length by about n times).

$$\begin{aligned} \text{Dec}([\![m_1]\!] \oplus [\![m_2]\!]) &= \text{Dec}(([\![m_1]\!] \times [\![m_2]\!]) \mod N^2) \\ &= (m_1 + m_2) \mod N \end{aligned} \tag{3.1}$$

Since it is possible to add encrypted numbers to each other, in a `for-loop`, an encrypted number can be added to itself d times to simulate a multiplication with d. However, note that the value d is in plaintext. Since addition is performed by doing multiplication on the encrypted numbers, we can, instead of a `for-loop`, take the encrypted value to the power of d to get this result, which can be implemented more efficiently than a `for-loop`. Furthermore, it has the advantage that it also works for $d < 0$. The case $d = -1$ is of special interest, because this makes subtraction of two encrypted numbers possible $(\text{Dec}([\![m_1 - m_2]\!]) = \text{Dec}([\![m_1]\!] \times [\![m_2]\!]^{-1} \mod N^2))$. The symbol $\otimes$ is used for such an operation on one plaintext number d and one encrypted number $[\![m_1]\!]$. Equation 3.2 shows how to calculate this multiplication with one encrypted number.

$$\begin{aligned} \text{Dec}([\![m_1]\!] \otimes d) &= \text{Dec}([\![m_1]\!]^d \mod N^2) \\ &= (m_1 \times d) \mod N \end{aligned} \tag{3.2}$$

While the Paillier PHE supports an efficient method to multiply an encrypted and a plaintext number, it does not support the multiplication of two encrypted numbers and is, therefore, not a fully homomorphic encryption scheme.

Algorithm 3.1: Paillier Create Keys

Parameters: Length of the modulus N in bit (b).
Result: The secure key (p, q) and the public key (N, g).

1 procedure `create`(**in** b)
2   **repeat**
3 p = random prime number with a length of $\frac{b}{2}$ bits
4 q = random prime number $\neq p$ with a length of $\frac{b}{2}$ bits
5 $N = p * q$
6   **until** `bitLength`$(N) = b$
7 $g = N + 1$
8   **return** (p, q) and (N, g)

Algorithm 3.2: Paillier Encrypt

Parameters: The plaintext message m ($< N$) that should be encrypted with the public key (N, g).
Result: Ciphertext c

1 procedure `encrypt`(**in** m, **in** N, **in** g)
2 r = random number that is smaller than N ($r < N$)
3 $c = g^m \cdot r^N \mod N^2$
4   **return** c

The Paillier HE is a probabilistic asymmetric encryption scheme like the well-known RSA scheme [66]. That means that an encryption can be performed by using the public key, which is derived from the secure (or private) key. However, for the task of decryption, the secure key is required, which cannot (efficiently) be calculated from the public key because this would require factoring a product of two large prime numbers. It is essential for the security of Paillier's cryptosystem (and also for RSA) that there is no known fast method for integer factorization of a product of two large prime numbers. The Algorithm 3.1 shows an example of a key generation function for the Paillier cryptosystem, which sets the generator g always to $N + 1$, as this allows a more efficient encryption function (see: [15]). Furthermore, the algorithm clearly shows that the secure key, which needs to be kept secret, contains the two large prime numbers p and q (e.g., 1024 bit long) and the public key contains the product N (e.g., 2048 bit long) of these two prime numbers.

Algorithm 3.2 and Algorithm 3.3 show the pseudocode for the encrypt and decrypt routines for the Paillier HE. The encryption algorithm also contains the obfuscating of an encrypted number with a random number r, which qualifies the Paillier HE as a probabilistic encryption scheme. This means that a specific plaintext message m can be represented by many possible ciphertexts $[\![m]\!]_1, [\![m]\!]_2, [\![m]\!]_3 \dots, [\![m]\!]_r$. The decryption with the right secure key will return the original message m for all the

---

Algorithm 3.3: Paillier Decrypt

Parameters: The ciphertext c that should be decrypted with the secure key (p, q) and the public key (N, g).

Result: Plaintext m

1 **procedure** decrypt(**in** c, **in** p, **in** q, **in** N, **in** g)

/* L- and H-function as defined in Paillier's work [58] at section 7 "Efficiency and Implementation Aspects" */

2   **procedure** L(**in** x, **in** y)

3   **return** $\frac{x-1}{y}$

4   **procedure** H(**in** x)

5   **return** L($g^{x-1} \mod x^2$, x)$^{-1} \mod x$

/* The Chinese Remainder Theorem */

6   **procedure** C(**in** m_p, **in** m_q)

7   **return** $m_p + (((m_q - m_p) \cdot (p^{-1} \mod q) \mod q) \cdot p)$

8 $h_p =$ H(p)   // can be pre-computed

9 $h_q =$ H(q)   // can be pre-computed

10 $m_p =$ L($c^{p-1} \mod p^2$, p) $\cdot h_p \mod p$

11 $m_q =$ L($c^{q-1} \mod q^2$, q) $\cdot h_q \mod q$

12 $m =$ C(m_p, m_q)

13   **return** m

---

possible ciphertext representations. While this is not required for correct homomorphic calculations with Paillier (imagine $r = 1$), it is important for the semantic security against chosen-plaintext attacks (IND-CPA) that Paillier's cryptosystem provides [58, 81]. Without this obfuscating, it would be possible to decrypt datasets without knowing the secure key because an attacker would only need to encrypt all possible plaintext values with the public key, store the plaintext and the corresponding ciphertext values in pairs, and compare the values of the encrypted dataset with the self-encrypted values for which the correct decryption is known. For datasets with a limited number of possible values like the voxel values of a volume, which usually contains no more than $2^{10} = 1024$ different values, this would be a trivial task.

Encrypted Rendering

The first step of the introduced privacy preserving rendering system is the encryption of the volume dataset (Figure 4.1 Acquisition Device). During the encryption stage, every single scalar voxel value of a volume dataset needs to be encrypted with Paillier's approach (see Algorithm 3.2). Meta data of the volume such as width, height, depth and the storage order of voxels will not be encrypted. The next step is to upload the encrypted volume dataset to a server (Figure 4.1 arrow from Acquisition Device to Cloud Server). For our approach, the device that encrypts the volume and uploads it to a server does not even need the secure key, because for encryption, only the public key is required.

When a rendered image is requested to be shown on a client, the client sends a rendering request to the server, which has the encrypted volume dataset (Figure 4.1 arrow from Client to Cloud Server). The rendering request contains further information about the settings of the rendering pipeline, such as the camera position, view projection, and (depending on the selected rendering type) also information about the transfer function that should be used. After the server receives such a rendering request, it uses the included pipeline settings and the already stored encrypted volume dataset to render the requested image (Figure 4.1 the rendering pipeline stages of the Cloud Server). To preserve privacy, the server does not have the secure key and can not, therefore, decrypt the volume data. The operations that are used for rendering an image from an encrypted volume dataset are limited to the homomorphic operations add ($\oplus$, Equation 3.1) and multiply with plaintext ($\otimes$, Equation 3.2), which are defined for Paillier's encryption scheme. When the rendering is finished, the server will send the calculated image data to the client (Figure 4.1 arrow from Cloud Server to Client). The resulting image that the client receives is still encrypted. Decrypting such an image is only possible for a client that knows the correct secure key. For everyone else, the image will be random noise (shown in Supplementary Video Material). Since every single pixel value is an encrypted number, every single pixel can be decrypted independently of the other pixels. For a
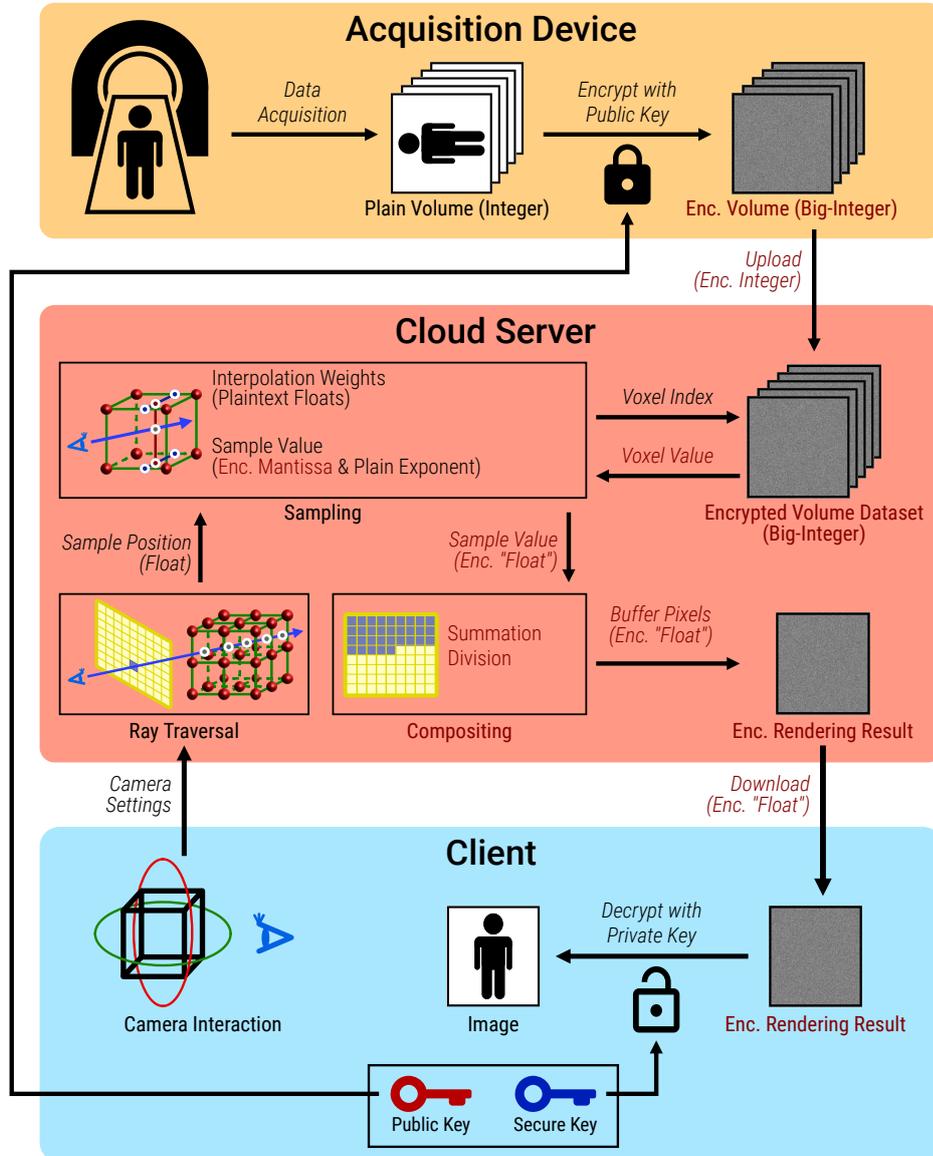
Figure 4.1: Our approach consists of a computer that produces, encrypts, and sends volume data to a server, which then renders the data and sends the result to a client. The client decrypts and visualizes the result. The text that belongs to encrypted data or processing is stated in red.

gray-scale image, that means one number per pixel. An RGB colored image requires three values that need to be decrypted per pixel.

In Section 4.1, we explain how the homomorphic operations of Paillier's HE can be used for X-ray sample integration. Furthermore, we will show how to use Paillier's cryptosystem with floating-point numbers, which allows us to perform trilinear interpolation. Chapter 5 explains a more advanced approach that allows the emphasizing of different density ranges in the rendered images.

4.1   Encrypted X-Ray Rendering

Ray-casting [39] is the most frequently used approach for volume rendering. Furthermore, ray-casting based algorithms can be easily and efficiently parallelized and can be implemented with fewer memory reads than slicing-based algorithms. Memory access is time-consuming, especially if every number that needs to be read is thousands of bits long. Therefore, we implement our privacy-preserving volume rendering approach with ray-casting. However, other direct volume rendering approaches developed for unencrypted data, such as slicing, can be used as well. Slicing on the server can be built by the same encrypted rendering pipeline components (sampling / interpolation, color mapping, compositing), which we will explain anon. Slicing could also be used to just perform the sampling on the server, transfer the slices to the client, and perform the compositing there. However, this would not fulfill our requirements because of the required network bandwith and the high computational requirement on the client.

The ray casting algorithm first calculates a viewing ray for every pixel of the final image (Figure 1.1 Ray Traversal - stage of the Server). These viewing rays will be calculated based on the camera position, up vector, opening angle, image resolution, and pixel index. At discrete and equidistant steps along the ray, the data of the volume is sampled (Figure 1.1 Sampling - stage of the Server). The last step is the compositing, where the final pixel value is calculated based on the sample values of a viewing ray (Figure 1.1 Compositing - stage of the Server).

X-ray rendering is a volume rendering approach where the sample value is mapped to a white color with monotonically increasing opacity, and the compositing is a summation followed by a normalization at the end of the ray traversal. If the sampling of the voxel values is done by nearest-neighbor filtering, the sum along a viewing ray can be calculated by only using the homomorphic add operation ($\oplus$) which is already defined for Paillier's cryptosystem (see Equation 3.1). The final normalization of all samples along a view ray cannot be done directly by the homomorphic operations of Paillier's encryption scheme because this requires a division that can result in a non-integer value that is not supported. However, the server could send the encrypted sum together with the sample count to the client, which can perform the division after decrypting the sum.

To improve the nearest-neighbor sampling with trilinear interpolation, a mechanism that allows the summing and normalization of encrypted values ($[\![m_1]\!], [\![m_1]\!]$), which are scaled

(a) Nearest Neighbor　　　　　　　　(b) Trilinear Interpolation
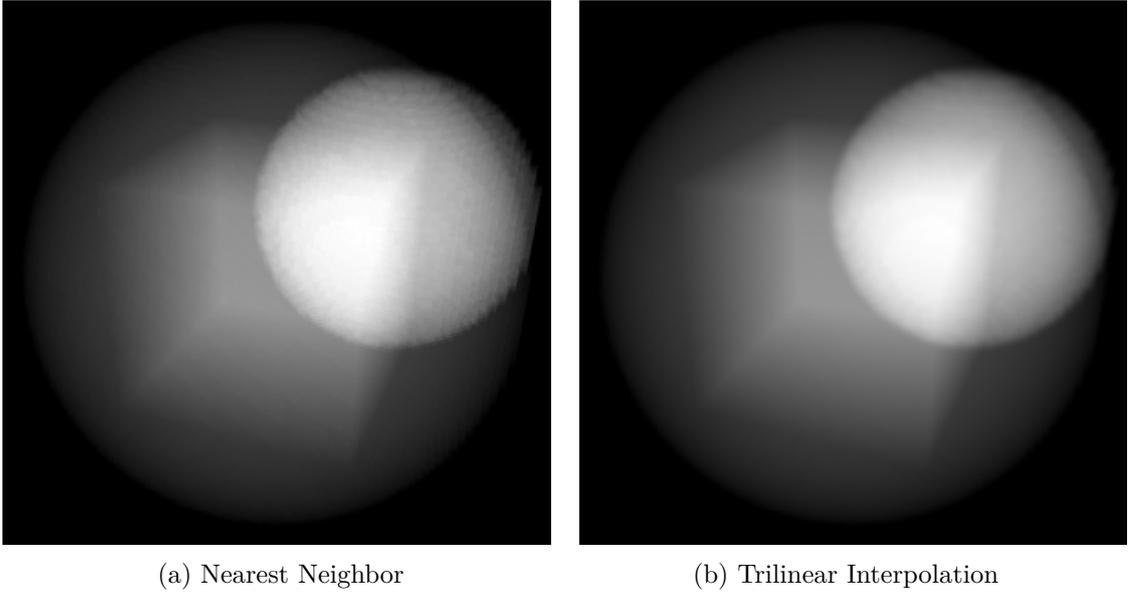
Figure 4.2: Results from encrypted X-ray rendering showing nearest neighbor (a) and Trilinear interpolation (b), which we also support.

by some plaintext weights (α_1, α_2), is required. For plaintext integers, the interpolation could be implemented around the integer arithmetic operations add, multiply, and divide (1D example: $(m_1 \cdot \alpha_1 + m_2 \cdot \alpha_2)/(\alpha_1 + \alpha_2)$). Since an arbitrary division is not supported by Paillier's cryptosystem, this is not directly feasible on encrypted data. A possible solution could be to use fraction types, which has an encrypted denominator and a plaintext numerator for storage and calculations. After the image is rendered, which contains such fractions as pixel values, the client can download it, decrypt the denominators, and perform the deferred divisions[1]. However, we decided to use a floating-point encoding, which is easier to implement and allows a shader code development as is usual for hardware accelerated rendering. With a floating-point representation of encrypted values, it is possible to multiply the eight neighboring voxels of a sample position with the distances between the samples and voxel position. These distances, which have a sum of 1.0, are the weights of the interpolation (1D example: $m_1 \cdot \alpha_1 + m_2 \cdot \alpha_2$). A floating-point encoding will also make the final division of the sample sum for X-ray rendering on the server side possible. While a floating-point encoding does not directly enable divisions in the encrypted domain, it can be used to approximate a division by a multiplication with the reciprocal of the divisor, as shown in Equation 4.1.

$$\frac{\sum}{n} \approx \text{Dec}\left([\![\textstyle\sum]\!] \otimes \left\lfloor \frac{1}{n} \cdot 10^\gamma \right\rfloor \right) \cdot 10^{-\gamma} \tag{4.1}$$

---

[1]   If the rendering pipeline is designed in a very static way, it is theoretically possible to know the final numerator upfront and let the client perform the required division without explicitly specifying the numerator. However, this is very inflexible, error prone, and requires an update for the client whenever a change on the server leads to a change of the final numerator.

The sum of samples along a viewing ray is denoted as $\sum$, and n is the count of samples. The precision of the approximation is defined by the count of decimal digits γ (e.g., $\gamma = 3$ for thousandth). Before the reciprocal of n is multiplied with $\sum$, the comma is moved γ digits to the right ($\cdot 10^\gamma$) and then rounded ($\lfloor \rfloor$). The multiplication with $10^{-\gamma}$, which moves the comma back to the correct position, can be achieved by subtracting γ from the exponent of the floating-point encoded result. Since the Paillier cryptosystem is defined over $\mathbb{Z}_N$, the result is only correct if no intermediate result is greater than $N-1$.

We will discuss the used floating-point encoding in Section 4.1.1. Figure 4.2 shows two images that were rendered from an encrypted floating-point encoded dataset. For the rendering of the left image, a nearest-neighbor sampling was used, and for the right image, a trilinear interpolation was used. The used dataset contains three objects with different densities: a solid cube in the center wrapped inside a sphere and another sphere at the top left front corner. The same dataset is also used for renderings shown in Figure 5.2 and Figure 5.3.

4.1.1   Encrypted Floating-Point Numbers

A floating-point number is defined as $m \cdot b^e$, where m is called the mantissa. The exponent e defines the position of the comma in the final number. The base b is a constant that is defined upfront (e.g., during the compilation of the application). We used a decimal system for convenience; therefore, our prototype uses $b = 10$. However, b can be any positive integer that is greater or equal to 2.

To calculate with floating-point arithmetic in the encrypted domain, we have chosen to use the approach developed for Google's Encrypted BigQuery Client [19]. The idea is to store the mantissa m and the exponent e of a floating-point number in two different integer variables. During the encryption of the floating-point number (m, e), only the mantissa m is encrypted using Paillier's cryptosystem. The exponent e remains unencrypted, which results in the floating-point number $(\llbracket m \rrbracket, e)$. This floating-point number representation is also used by the *python-paillier* library [7], the Java library *javallier* [7] and in the work by Ziad et al. [83].

For an addition of two such encrypted floating-point numbers, both need to have the same exponent. Therefore, the exponents of both numbers must be made equal before the actual addition, if they are not already equal. Hence, it is not possible to increase the exponent if the mantissa is encrypted because that would require a homomorphic division of the encrypted mantissa, which is not possible. Therefore, the floating-point number with the greater exponent needs to be changed. On the other hand, decreasing the exponent of a floating-point number is not a problem because it requires a homomorphic multiplication of the encrypted mantissa with a plaintext number, which is possible with Paillier. Equation 4.2 shows how to calculate the new mantissa $\llbracket m_n \rrbracket$ that is required for decreasing the exponent of the floating-point number $(\llbracket m_o \rrbracket, e_o)$ to the lower exponent e_n. The new floating-point number is defined as $(\llbracket m_n \rrbracket, e_n)$, which represents exactly the

---

Algorithm 4.1: Paillier Floating-Point Add

Parameters: Encrypted mantissas $[\![m_1]\!], [\![m_2]\!]$ and plaintext exponents e_1, e_2 of the two floating-point numbers that should be summed. b is the used base, e.g. 10 for a decimal system.

Result: Encrypt mantissa $[\![m_s]\!]$ and plaintext exponent e_n.

1 procedure fpAdd(**in** $[\![m_1]\!]$, **in** e_1, **in** $[\![m_2]\!]$, **in** e_2)

2   **if** $e_1 > e_2$ **then**

3 $[\![m_1]\!] = [\![m_1]\!] \otimes b^{e_1 - e_2}$

4 $e_n = e_2$

5   **else if** $e_1 < e_2$ **then**

6 $[\![m_2]\!] = [\![m_2]\!] \otimes b^{e_2 - e_1}$

7 $e_n = e_1$

8   **else**

9 $e_n = e_1$

10   **end**

11 $[\![m_s]\!] = [\![m_1]\!] \oplus [\![m_2]\!]$

12   **return** $\{[\![m_s]\!], e_n\}$

---

same number as $([\![m_o]\!], e_o)$. It is just another way to store it.

$$[\![m_n]\!] = [\![m_o]\!] \otimes b^{e_o - e_n} \tag{4.2}$$

When both floating-point numbers $([\![m_1]\!], e_1)$ and $([\![m_2]\!], e_2)$ have the same exponent $e_1 = e_2 = e_n$, the homomorphic sum $[\![m_s]\!]$ of both mantissas can be calculated by the add operation defined for Paillier (see Equation 3.1), which results in the final floating-point number $([\![m_s]\!], e_n)$. The Algorithm 4.1 shows this approach for summing two floating-point numbers with encrypted mantissas. The lines from 2 to 10 bring the exponents of both floating-point numbers to the same value (e_n), and line number 11 contains the addition of the encrypted mantissas.

A multiplication with a floating-point number that contains an encrypted mantissa $([\![m_1]\!], e_1)$ and a floating-point number with a plaintext mantissa (m_2, e_2) can be achieved by multiplying the mantissas with the multiplication operation defined for Paillier ($[\![m_n]\!] = [\![m_1]\!] \otimes m_2$, Equation 3.2) and a plaintext addition of the exponents ($e_n = e_1 + e_2$). This is also stated in line 10 and 11 of the Algorithm 4.2, which is sufficient for a correct result. The lines from 2 to 9 contain a performance optimization, which prevents the intermediate result of $[\![m_e]\!]^{m_d}$, which is computed before $\bmod N^2$ is applied in line 10, from being unnecessarily large (see Equation 3.2). This optimization is also used by the python library *python-paillier* [7] in `paillier.py` and the java library *javallier* [6] in `PaillierContext.java`.

Signed numbers can be represented by using a two's complement representation for the mantissa m. The exponent e does not change. If v is a negative integer, the two's

---

Algorithm 4.2: Paillier Floating-Point Multiply

Parameters: Encrypted mantissa $[\![m_1]\!]$, plaintext mantissa m_2 and the plaintext
exponents (e_1, e_2) of the two floating-point numbers that should be
multiplied.
N is the modulus of the used public key.

Result: Encrypt mantissa $[\![m_n]\!]$ and plaintext exponent e_n.

1 **procedure** fpMultiply(**in** $[\![m_1]\!]$, **in** e_1, **in** m_2, **in** e_2)
2     $m_n = N - m_2$ // negative of m_2
3   **if** $m_n \leq$ *max. value that can be encrypted by current* N **then**
        // If the plaintext is large, exponentiate using its negative instead.
4         $[\![m_e]\!] = [\![m_1]\!]^{-1} \mod N^2$ // multiplicative inverse of $[\![m_1]\!]$ in the integers
        modulo N^2
5         $m_d = m_n$
6   **else**
7         $[\![m_e]\!] = [\![m_1]\!]$
8         $m_d = m_2$
9   **end**
10     $[\![m_n]\!] = [\![m_e]\!] \otimes m_d$
11     $e_n = e_1 + e_2$
12   **return** $\{[\![m_n]\!], e_n\}$

---

complement in the integer modulo N can be calculated by: $m = v + N$. In the encrypted domain, the additive inverse $-m$ of m is defined by the multiplicitive inverse $[\![m]\!]^{-1} = [\![i]\!]$ of $[\![m]\!]$ in the integers, modulo N^2 ($[\![i]\!]$ is defined by: $[\![m]\!] \cdot [\![i]\!] = 1 \mod N^2$ and can be computed from $[\![m]\!]$ and N^2 by the *extended Euclidian algorithm* [36]). This complement representation for encrypted numbers can also be used for a subtraction of two encrypted numbers. Since, the first operand of a subtraction can be added to the additive inverse of the second operand ($\mathrm{Dec}([\![m_1 - m_2]\!]) = \mathrm{Dec}([\![m_1]\!] \times [\![m_2]\!]^{-1} \mod N^2)$).

With the floating-point encoding explained in this section, it is possible to perform a trilinear interpolation of voxel values because the encrypted voxel values can be multiplied by the fractional distances between a sample position on a viewing ray and the actual voxel position. Furthermore, divisions of an encrypted number ($[\![m]\!], e$) by a plaintext number d can be approximated by a multiplication of the encrypted number ($[\![m]\!], e$) with the reciprocal ($\lfloor 1/d \cdot 10^\gamma \rfloor, -\gamma$) of d (γ defines the precision - compare with Equation 4.1).

Transfer Function

In this chapter, we discuss the challenges of building a transfer function approach that works for a probabilistic PHE scheme, and we show a novel and practical solution for a simplified transfer function. It is not possible to use the transferred values for an alpha blending sample compositing because this would require a multiplication of two encrypted values, which is not possible with Paillier's cryptosystem. However, the transfer function can be used to highlight specific density ranges at X-ray rendering, which helps an observer to distinguish between different objects inside a volume.

A transfer function for non-encrypted voxel values can be implemented as an array with the possible voxel values as indices and the assigned color as values of the array. The evaluation of such a transfer function is as simple as reading the value from the array at the index, which is equal to the voxel value that should be mapped. However, this cannot be efficiently implemented for encrypted data. For non-encrypted voxel values, such a transfer function array will have a length that is equal to the amount of possible voxel values, which is only $2^8 = 256$ for 8-bit voxels or $2^{10} = 1024$ for 10-bit voxels. An encrypted volume dataset will probably not contain two equal voxel values, because of the obfuscation during the encryption. That means an encrypted dataset will probably have as many different voxel values as it has voxels. Therefore, an array as transfer function will not work because it would be at least as big as the volume itself.

Another approach for non-encrypted data is to store just some supporting points that contain the density and color. The evaluation for this transfer function approach is achieved by interpolating the color between the value of the next lower and next greater supporting point. To find the neighboring supporting points of the voxel value that should be transferred, comparison operators such as lower than ($<$) or greater than ($>$) are required. However, comparison operators cannot exist for probabilistic PHE schemes like Paillier because that would break its security (see Section 8.4). Therefore, the question is how to implement a function $f : X \to Y$ that can map finite sets of numbers X to another set of numbers Y by just using the operations *add* ($\oplus$) and *multiply with constant*

($\otimes$). The result of this function is again an encrypted number. A promising approach that can achieve this was presented by Wamser et al. [77] in their work on "oblivious lookup-tables".

5.1   Oblivious Lookup Tables

Let $X = \{x_1, x_2, ..., x_n\}$ be an enumeration of values that should be mapped to $Y = \{y_1, y_2, ..., y_n\}$ by the lookup function $f(x_i) = y_1$. The idea is to create a vector $\vec{v_i}$ for every $x_i \in X$ with the same cardinality as X ($|\vec{v_i}| = |X|$) and define the evaluation of a lookup by the dot product shown in Equation 5.1.

$$\vec{v_i} \cdot \vec{l} = y_i \tag{5.1}$$

The scalar value y_i is the result of the lookup. For a transfer function, this would be the value of one color channel. The vector $\vec{l}$ can be calculated form the linear Equation 5.2.

$$V \cdot \vec{l} = \vec{y} \tag{5.2}$$

V is a square matrix of full rank with $n = |X|$, that uses all vectors $\vec{v_i}$ as rows. However, this linear equation needs to be solved only once. Therefore, the client can calculate $\vec{l}$ upfront based on unencrypted numbers. The Equation 5.2 has a unique solution, if all vectors $\vec{v_i}$ are linearly independent. Hence, the crucial part is to find an approach to extrapolate every vector $\vec{v_i}$ only from one single x_i so that the $\vec{v_i}$ are linearly independent from each other. Wamser et al. [77] suggest to use a Vandermonde-Matrix as V (Equation 5.3), because it fulfills these requirements.

$$V = \begin{pmatrix} 1 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n^1 & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix} \tag{5.3}$$

From the creation rule of the Vandermonde-Matrix, it follows that a $\vec{v_i}$, which is equal to the i-th row of the matrix V, is defined as $\vec{v_i} = (1, x_i^1, x_i^2, \cdots, x_i^{n-1})$. The lookup function $f(x_i)$ can, therefore, be stated as:

$$f(x_i) = (1, x_i^1, x_i^2, \cdots, x_i^{n-1}) \cdot \vec{l} = y_i \tag{5.4}$$

The dot product in Equation 5.4 (and Equation 5.1) can be calculated even if $\vec{v_i} = (1, x_i^1, x_i^2, \cdots, x_i^{n-1})$ is encrypted because only the operations *add* ($\oplus$) and *multiply* ($\otimes$) that are defined for the Paillier HE are required for calculating a dot product. However, it is not possible to calculate the vector $\vec{v_i}$ from an encrypted $[\![x_i]\!]$, because this would involve multiplications of two encrypted numbers, which is not possible with Paillier. A theoretical solution for this could be to store the vector $\vec{v_i}$ instead of scalar x_i as the value of a voxel. For a volume dataset, where the voxel values have only a resolution of

8 bits, this would lead to a vector length of $n = 2^8 = 256$. Therefore, the required storage size for the volume will increase 256 times.

During our research I developed and tested alternative matrix creation schemes that can be used instead of the Vandermonde-Matrix. For example, I was able to create a scheme that can be calculated much more efficiently than the exponential calculation for the Vandermonde-Matrix by taking advantage of the properties of a transfer function. Some of the more useful and interesting matrix creation schemes are stated in Appendix B. However, none of this schemes can help with the fundamental problem of the storage overhead, since they cannot be computed in the encrypted domain.

A volume with $512 \times 512 \times 512$ voxels and a resolution of 8 bits per voxel requires $512^3 \cdot 8$ bits/8 bits $= 134,217,728$ Bytes $= 128$ MB. The same volume encrypted by Paillier HE with a public key length that can be considered as secure (2048 bits) requires $512^3 \cdot 2 \cdot 2048$bits/8bits $= 64$ GB. If the scalar voxel values x_i are replaced by the vectors $\vec{v_i}$ with a length of 256, the volume will require 64 GB $\cdot 256 = 16$ TB. While a volume dataset with 16 Terabyte is probably better than a transfer function that is at least as big as the encrypted volume, the overhead in terms of storage and computation is still too big to be practical. Therefore, we develop a simplified and novel transfer function approach with a considerably lower storage overhead, which we discuss in the next two sections.

5.2 Density Range Emphasizing

Our simplified transfer function approach is based on the observation that it is possible to compute the dot product of a vector with encrypted values and a vector with plaintext values. Furthermore, the dot product can be used to calculate an encrypted scalar value indicating the similarity of an encrypted vector and a plaintext vector. This will work if both vectors have length 1. Therefore, our approach is to encode the density values of each voxel as a vector and encrypt each component of this vector by the Paillier encryption algorithm (see Algorithm 3.2). In order to highlight a user-defined density range, the density value at the center of this range needs to be encoded as a vector. Note that this vector is not encrypted. The encrypted volume rendering engine can now compute the dot product between this vector and the encrypted vector of a sample position. Then the ray-casting algorithm needs to sum up the results of the dot products along a ray instead of the density values. This approach allows a user to emphasize a selectable density range in the rendered image. Figure 5.2 contains images that were created using this approach. The top left subfigure shows a result of an X-ray rendering for comparison. All other subfigures show results for different density ranges that are emphasized. The density that is encoded as vector that was used for the dot-product calculation is specified in the caption of each sub figure.

The density-to-vector encoding scheme we used is based on an HSV-to-RGB color conversion. The exact encoding scheme is stated in Algorithm 5.1. Figure 5.1 illustrates the magnitude of the vector components for all possible density values. Furthermore, the

---

Algorithm 5.1: Encode Density

Parameters: The normalized density that should be encoded as a vector with dim dimensions.

Result: Vector v

1 procedure encodeDensity(**in** density, **in** dim)

2   initialize vector v with length dim and set all indices to 0

3 $s = \mathsf{density} \cdot 2 \cdot (\mathsf{dim} - 1)$

4 $f = (\lfloor s \rfloor + 1)/2$

5 $d = \lfloor f \rfloor$

6 $\mathsf{v}[d] = 1$

7   **if** $d > 0$ **and** $d = f$ **then**

         // First half of the density range where index d is 1

8 $\mathsf{v}[d-1] = 1 - (s - \lfloor s \rfloor)$

9   **else if** $d + 1 < \mathsf{dim}$ **and** $d < f$ **then**

         // Second half of the density range where index d is 1

10 $\mathsf{v}[d+1] = s - \lfloor s \rfloor$

11   **return** normalize (v)

---

response intensities for user-defined emphasizing densities at 0.45 and 0.85 are shown. At the last line of Algorithm 5.1, the calculated vector is normalized. This is important to make sure that the result of the dot product is always between 0 and 1 and to ensure that the highest possible dot product result (1) is at the user-defined emphasizing density.

There are other and possibly better density-to-vector encoding schemes. However, the HSV-based encoding leads to results that feel natural, especially while smoothly increasing or decreasing the emphasizing density. The encoding scheme should in any case be chosen in such a way that the curve created by the dot product is steep and narrow (see dashed lines in Figure 5.1), so that the density selected by the user can be seen as clearly as possible in the resulting image. The Algorithm 5.1 takes not only the density that should be encoded as parameter, but also the count of dimensions of the returned vector. Increasing the count of dimension not only makes the dot product response curve more steep (See Figure 5.1 and compare the dashed lines in the left and right plot of the second row.), but also increases the required storage size of the encoded and encrypted volume dataset. Note that the count of dimensions must be the same during the encryption of the volume and for the encoding of the user-defined emphasizing density. This also means that the amount of computations required for the volume rendering depends on the number of dimensions used for encoding the volume.
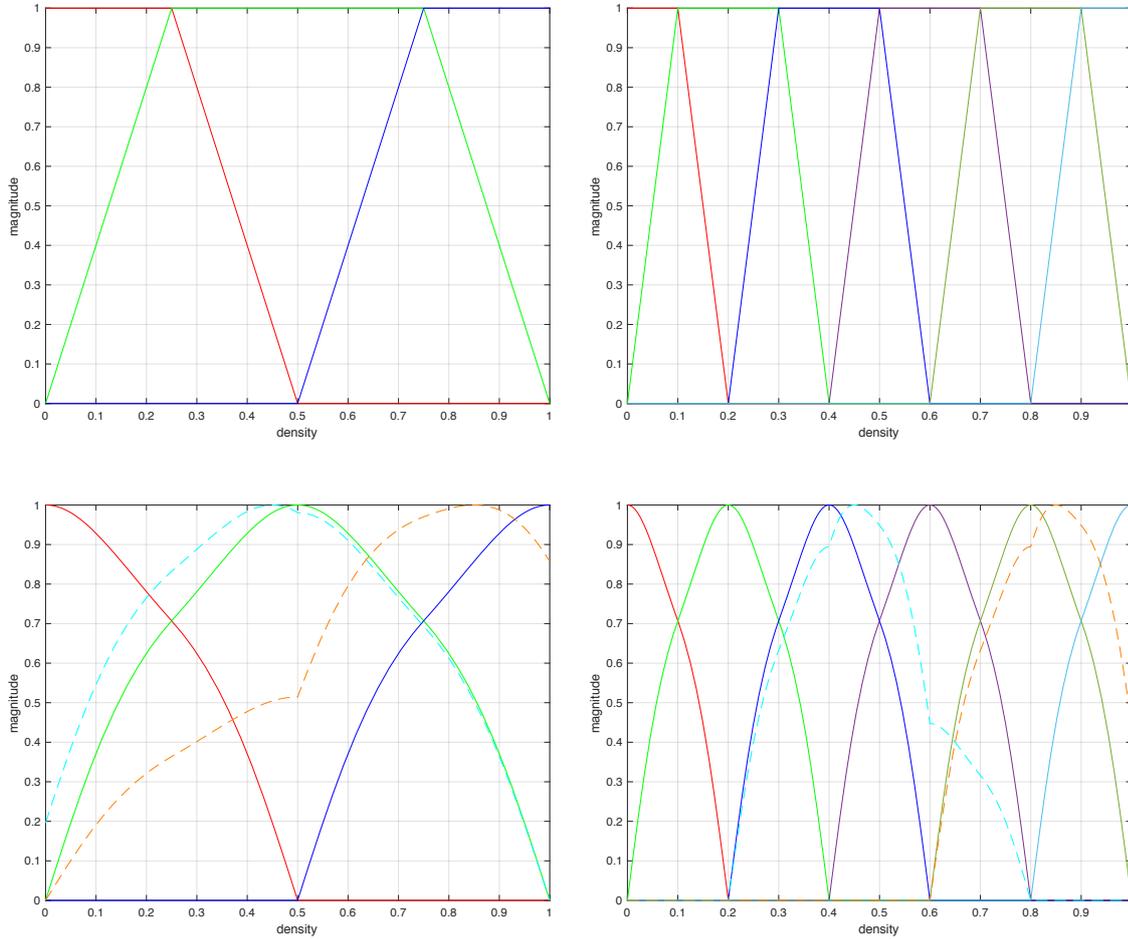
Figure 5.1: Visualization of density encoded as vectors. The left column shows an encoding in 3 dimensions and the right column an encoding in 6 dimensions. The scalar value (density) of the voxel is represented on the x-axes. The magnitude of each vector component at a specific density is represented by the curves. The first component is drawn in red, the second in green, red, purple, olive and light blue. The first row illustrates the vector before normalization. At the second row the normalized vectors are illustrated. The dashed curves shows the result of the dot product between the encoded voxel value and a *TF-Node* vector for a density of 0.45 in cyan and a density of 0.85 in orange.
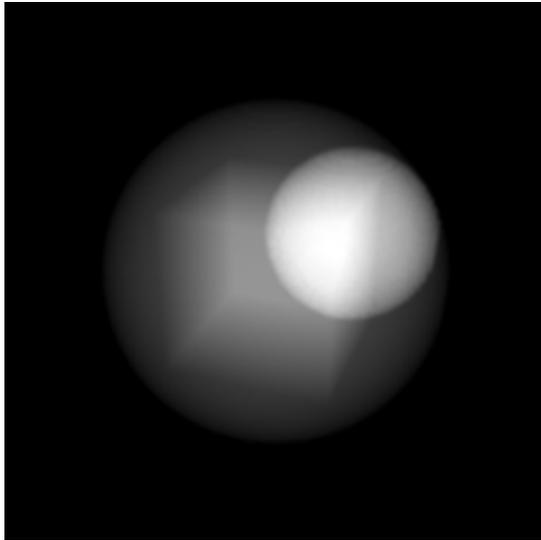
(a) X-ray
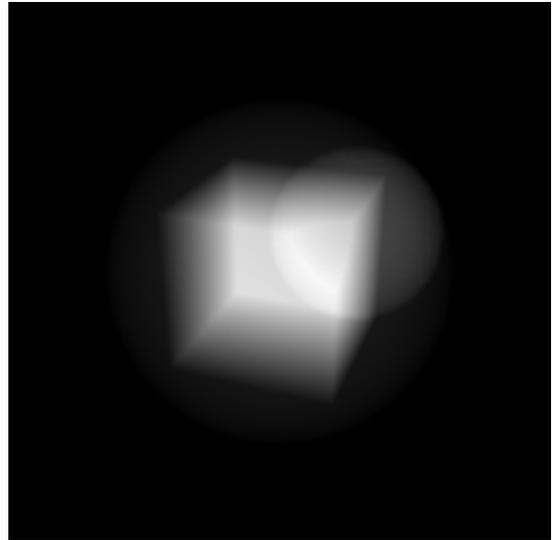
(b) emphasized density: 0.653
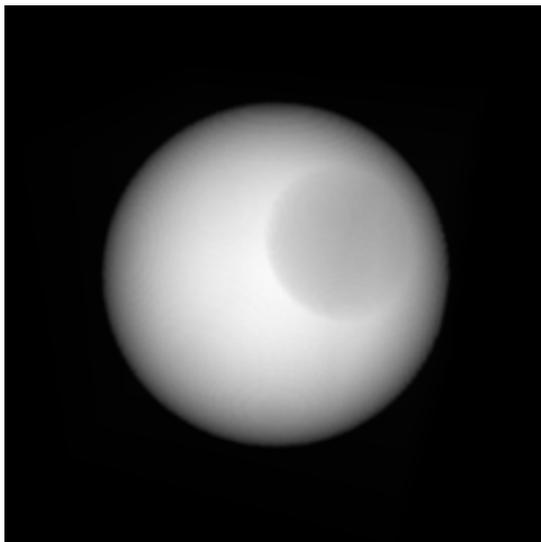
(c) emphasized density: 0.331
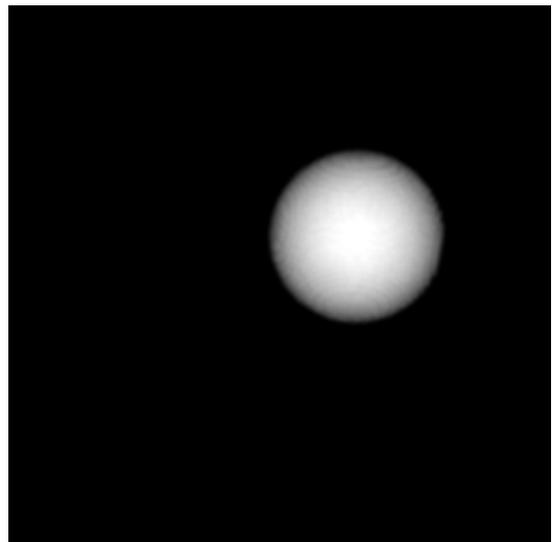
(d) emphasized density: 0.781

Figure 5.2: First image shows an X-ray rendering result for comparison with the other three images that are created by our encrypted density emphasizing approach. The volume density values are encoded with 4-dimensional vectors.

5.3 Simplified Transfer Function

It is possible to add RGB colors to the rendered images based on the density range emphasizing described in the last section. This is useful because RGB colors allow a user to emphasize different densities in the same image while keeping the densities distinguishable (see Figure 5.3). Since the dot product between an encoded and encrypted voxel value and a user-defined encoded density is an encrypted scalar value, a multiplication with another plaintext number is possible. For our simplified transfer function approach, the dot product result needs to be multiplied with a user-defined RGB color vector. As the dot product expresses the similarity between the voxel value and the user-defined density, the intensity of the resulting RGB color will be high if the densities are similar, and low otherwise. Since the RGB color vector is not encrypted, the multiplication between the encrypted dot product result and the RGB color vector can be archived by three separate homomorphic multiplications ($\otimes$) of one encrypted and one plaintext number (see Equation 3.2). The result of such a multiplication is an encrypted RGB color. This calculation can be performed not only for one density-RGB-color-pair, but also for multiple such pairs. For a better understanding, we will call such a pair consisting of a density and an RGB color a *transfer function node (TF-Node)*.

Equation 5.5 shows the transformation for one encoded and encrypted voxel value $[\![\vec{v}]\!]$ to an encrypted RGB color $[\![\vec{c_v}]\!]$. The symbol $\bigoplus$ is used instead of $\sum$, because the sum of encrypted vectors needs to be calculated. The variable n denotes the count of user defined *TF-Nodes*. The vectors $\vec{d_i}$ and $\vec{c_i}$ are the encoded density and RGB color of the *TF-Node* with index i. The symbol $\odot$ is used as operator for a dot product between one encrypted vector and one plaintext vector.

$$[\![\vec{c_v}]\!] = \bigoplus_{i=0}^{n} \left([\![\vec{v}]\!] \odot \vec{d_i} \right) \otimes \vec{c_i} \tag{5.5}$$

To obtain the final encrypted RGB color of a pixel, the sum of all encrypted RGB sample values $[\![\vec{c_v}]\!]$ along a viewing ray needs to be calculated. The total RGB vector needs to be divided by the sample count as usual for averaging and, furthermore, by the count of *TF-Nodes*. This can be achieved by dividing each component of the total RGB vector by the product of the sample count and the count of *TF-Nodes*. The method to approximate a division of an encrypted number is stated in Equation 4.1. After calculating this for every image pixel, the entire encrypted image is sent to the client. A client that knows the right secure key can now decrypt each RGB component of each pixel and display the colored image. Example images rendered with this approach are shown in Figure 5.3, Figure 7.3 and Figure 7.2.
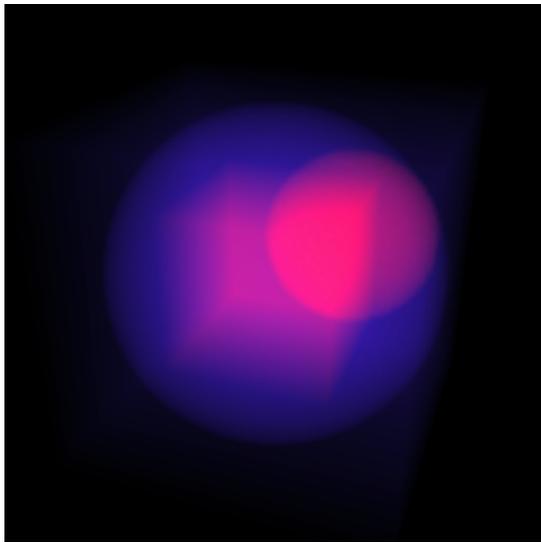
(a) blue at 0.279, red at 0.797
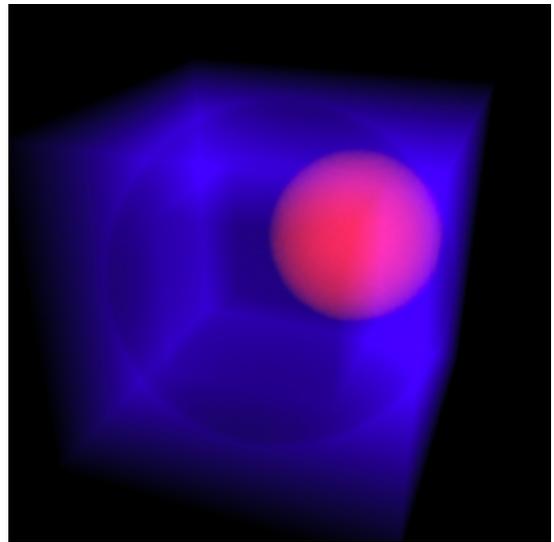
(b) blue at 0.000, red at 1.000
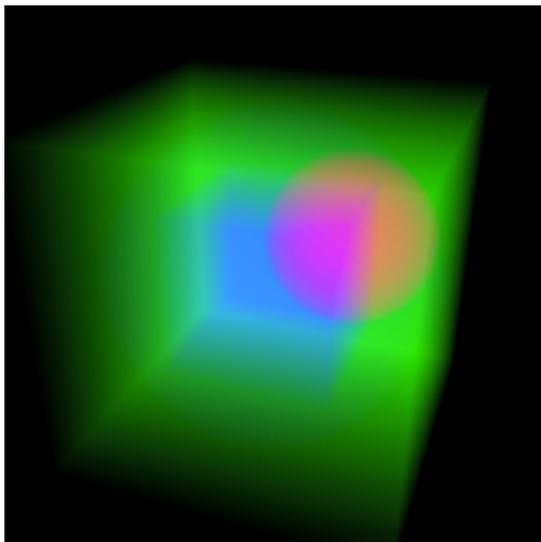
(c) green at 0.076, blue at 0.651, red at 1.000
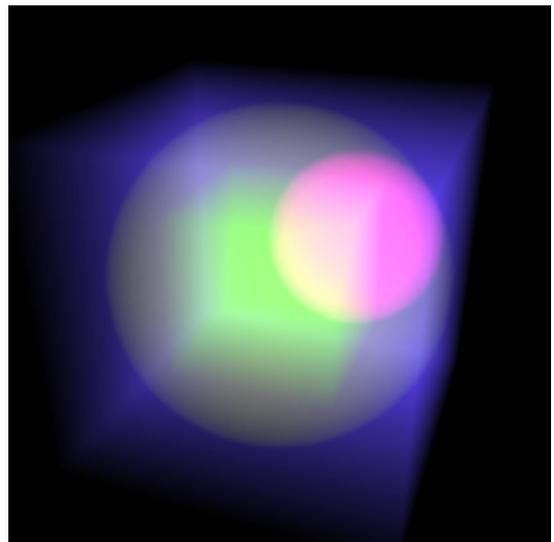
(d) blue at 0.000, yellow at 0.293, green at 0.664, purple at 1.000

Figure 5.3: Images are created by our simplified transfer function approach. The volume data voxel values are encoded by four-dimensional vectors. The subfigures shows results of different transfer functions applied to the same encrypted dataset.

GPU Implementation

In this chapter we want to discuss the implementation of our homomorphic encrypted X-ray volume rendering for Graphics Processing Units (GPU). It was the goal of this part to investigate the performance potential of a GPU in the context of volume rendering with big-integers that are at least a magnitude larger than the common maximal native integer machine word size of 24 or 32 bit supported by GPUs. Since there was no ready to use platform and GPU type independent big-integer libraries that we could use, we first need to implement our own big-integer library for GPUs. Our library is based on Vulkan [75], because Vulkan is a modern and fast API for GPUs and other accelerator hardware. It has a broad platform and vendor support and, hence, is a future proof choice. Furthermore, a big-integer library for Vulkan is something truly new, since we were not able to find a big-integer library designed for Vulkan. While Vulkan has the capability to use a GPU only for general-purpose computing by using compute shaders it also features a classical GPU rendering pipeline with a vertex shader, rasterizer and a fragment shader. This is a big differentiator when compared to CUDA (where nearly all big-integer libraries for GPUs are implemented with) or OpenCL. The classical graphics pipeline allows us to use the highly optimized texture units for accessing our encrypted volume data and do not need to design our own algorithm for loading voxels from 3D textures for rendering. Letting the GPU driver optimize the memory access is not only advantageous, because of the reduced workload for us but also since it allows a more direct comparison with a classical GPU based ray casting approach, since the efficient usage of the available memory bandwidth often has a greater impact on the overall performance of a procedure running on the GPU than the actual calculations.

At a high level view, the GPU implementation of the homomorphic encrypted X-ray rendering works like the ray casting algorithm for GPUs as described by Krüger and Westermann [39] (see Section 2.1.6). So first the encrypted volume is stored in multiple 3D textures on the graphics memory. Then the front face and back face texture which holds the positions, where the viewing ray of a screen pixel enters and exits the volume,

are rendered. The difference between the 3D vectors stored at the same pixel position in the back face and the front face texture then represent the viewing ray direction for each screen pixel. In the second render pass the voxels of the volume dataset are sampled along the viewing ray of each screen pixel. For homomorphic encrypted X-ray rendering the samples are encrypted values represented as big-integers that need to be assembled from multiple 3D textures. The accumulation along a ray is done by the addition operation defined for Paillier's cryptosystem which basically means multiplying two big-integer values and reducing the product by calculating modulo the squared public key modulus (N^2). The final result of a viewing ray, which is also a big-integer, is then written to the target frame buffer that contains multiple *Color Attachments* (basically 2D textures).

The library that we will introduce in the chapter consists of two parts, the host code which runs on the CPU and the device code which runs on the GPU. The host code does not only contain the code that is required for controlling the device but also has a full featured big-integer library. The big-integer library on the host is required for tasks like converting number representation, creating Paillier keys on the one hand, and as a testing and reference implementation on the other hand. Further details on the purpose of the host version of the big-integer library are stated in Section 6.4.

The host code for the GPU big-integer project is implemented in C++, while the code that runs on the device (GPU) is implemented in GLSL. The project is built with CMake [34] which makes it cross platform compatible. It should theoretically be buildable on all platforms that are supported by CMake and for which a Vulkan software development kit (SDK) [44] exists. So far, the support of Linux, Mac OS, and Windows has been confirmed by tests.

6.1 Magnitude Storage

The implemented big-integer library uses a *fixed radix number system (FRNS)* with 32 bit unsigned integers as words. Therefore, the radix is $2^{32} = 4,294,967,296$. Since the storage word type is defined by a precompiled constant and also all other values that depend on the radix, it is possible to use other word types such as 8, 16 and 64 bit unsigned integers. Especially smaller word types are useful for debugging because intermediate results can be recalculated more easily. Furthermore, the concept with the changeable radix allows performance comparisons for different word types. However, we will focus on 32 bit words, since all the runtimes shown in Section 7.2 are observed from big-integers with a 32 bit unsigned integer as base storage word type.

The fixed big-integer class uses an array of unsigned integers to store the magnitude of the number it represents. (The arbitrary long version uses a pointer of unsigned integers.) The words of the magnitude are stored in little-endian order. That means the least significant word is stored at the lowest index (0) or rather at the lowest memory address and the most significant word is stored at the last array index (array length − 1) or rather at the highest memory address. The alternative would be big-endian word order which stores the most significant word at lowest memory address which is more in line to

the human notation of numbers with Arabic numerals. None of the two word orders is significantly better than the other. Even with the existing big-integer libraries, there does not seem to be a clear preference for a word order. Javas `java.math.BigInteger` class [57] uses big-endian, Colin Plumb's C library [62] support big- and little-endian, TTMath [70] uses little-endian and GMP [16] always uses the native endianness of the hardware it is running on. Our library uses a little-endian word order, because it ensures that the words with the same significants (would be the same decimal place at decimal system) has the same array index for every big-integer. So the least significant word is always at the index 0, the word for the second radix is always at the array index 1, and so on. That means the words with the same significants can be aligned within different big-integers independent from the actual length of the big-integers. With a big-endian word order the word index with a specific significants must be calculated back from the end of the magnitude storage array. Therefore, the little-endian word order make the implementation for big-integers with an arbitrary size a bit easier.

6.2 Big-Integer Variable and Procedure Conventions

For an improved readability of this chapter and the big-integer related algorithms all variables that represent a big-integer variable will be denoted with a capital letter and single word values with a lowercase letters. Since the storage of a big-integer variable is effectively an array, we follow C programming language conventions and state the access of a word at index i within the big-integer A as $A[i]$ and not with a subscript as common in mathematics. Furthermore, the array index starts from 0.

There are two different length units for big-integers common and required. One that specifies the total length in bit and one that defines the word count. We will denote a bit length specification with the variable l and the size in words with the variable s. If 32 bit long words are used as the storage primitive, a l bit long integer will require $s = \lceil \frac{l}{32} \rceil$ words for storing it.

The big-integer algorithms stated in this chapter references some utility procedures. To avoid misunderstandings, we would like to briefly define them in the following list.

- `BigInteger(`**in** a`)`: Creates a new big-integer value that represents the same value as a. Hence, a is used as the least significant word of the new big-integer and all more significant bits of the new big-integer are set to zero.

- `bitLength(`**in** A`)`: Returns the length of the big-integer A in bit. Returns 0 if A is zero, hence A does not have a single bit that is 1.

- `wordLength(`**in** A`)`: Returns the size of the big-integer A in words. The procedure `wordLength` does return 1 also for big-integers which represent zero, therefore, `wordLength` will never return 0.

- `findLowestSetBit(`**in** A`)`: Returns the index of the least significant (rightmost) bit that is one at the big-integer A (the number of bits that are zero on the least significant (rightmost) side of A).

- `findLowestSetWord(`**in** A`)`: Returns the index of the least significant (rightmost) words that is non zero at the big-integer A (If 32 bit long words are used as the storage primitive returned value will be equal to $\lfloor \frac{\text{findLowestSetBit}(A)}{32} \rfloor$).

- `shiftLeft` A by n `bits`: Moves all bits in A n to the left (makes the bits more significant).

- `shiftRight` A by n `bits`: Moves all bits in A n to the right (makes the bits less significant).

- `addTwoWords(`**in** a, **in** b, **in** c, **out** r`)`: Sums up the two single word integers a and b (see Algorithm A.2). If the input carry flag c is true the procedure adds another 1 to the sum of a and b. This carry flag input parameter c is required when the procedure is used in routines that perform additions with a big-integer. Since every addition of two words can lead to a carry flag and this carry needs to be propagated to the addition of the next higher word. The sum of a, b, and c will be saved into the output parameter r. If the summation produces an overflow, because the length of the variable r is not sufficient for storing the sum, the procedure will return true in order to indicate that the addition has led to a carry.

- `addInt(`**in** a, **in** p, **inout** T, **in** n`)`: Adds a one word long integer a to the big-integer T starting at the index p (see Algorithm A.4). n is the length of T. A potential carry flag will be returned. If p is 0 the procedure will act like a normal addition of a big-integer and single word long integer.

- `addTwoInts(`**in** a'', **in** a', **in** p, **inout** T, **in** n`)`: Adds a two words long integer a (a' is the least significant word and a'' the most significant word) to the big-integer T starting at the index p (see Algorithm A.3). n is the length of T. A potential carry flag will be returned. If p is 0 the procedure will act like a normal addition between a big-integer and two word long integer.

- `mulTwoWords(`**in** a, **in** b, **out** r'', **out** r'`)`: Multiplies the two single word integers a and b and store the product into the two word long integer r (see Algorithm A.5). Whereby r' is the lower (least significant) and is r'' high (most significant) word of r.

- `mulInt(`**in** A, **in** k`)`: Multiplies the big-integer A with a single word integer k and return a new big-integer with the product (see Algorithm A.6).

6.3 Implemented Big-Integer Operations

While the library is designed around the concepts stated by Donald Knuth in the section *Classical Algorithms* of *The Art of Computer Programming* [35], lots of implementation

details are based on existing implementation, namely *TTMath* from Tomasz Sowa [70], Java `java.math.BigInteger` class [57] and Colin Plumb's C library [62]. We will continue with a list of the most important operations implemented into our big-integer library. However, not all of the listed procedures are implemented for the GPU, some are only available on the C++ code for the host. The procedures listed under Section 6.3.4, Section 6.3.5, and Section 6.3.6 are only available on the host (CPU) side.

6.3.1   Comparison Operators

The library contains comparison operators like the one that are usually found for integers in modern programming languages. All listed comparison operators expect two big-integers which should be compared as parameters. The procedures are implemented for the host and for the device.

- *lessThan* (<): Compares two big-integers and returns true if the first is smaller than the second, false otherwise.

- *lessThanOrEqualTo* (<=): Compares two big-integers and returns true if the first is smaller than the second or equal to it, false otherwise.

- *greaterThan* (>): Compares two big-integers and returns true if the first is greater than the second, false otherwise.

- *greaterThanOrEqualTo* (>=): Compares two big-integers and returns true if the first is greater than the second or equal to it, false otherwise.

- *equalTo* (==): Returns true only if the both input big-integers represent the same value, false otherwise.

- *notEqualTo* (!=): Returns true if the two input big-integers does not represent the same value and false if the two both big-integers represent the same value.

6.3.2   Bit Manipulation

These procedures work by making use of the fact that the words of the numbers are stored in binary form. These are useful for fast multiplications and divisions with a number in the form of 2^x, bit masking, and other shortcuts for arithmetic operations. The following list contains the operations that are implemented on the host and on the device version. However, the host implementation contains even more procedures for manipulation and checking at the bit level.

- *shiftLeft* («): Shifts the bits of a big-integer to the left for a specified count of bits.

- *shiftRight* (»): Shifts the bits of a big-integer to the right for a specified count of bits.

- *setBit*: Sets the bit at a specific position to 1.

- *clearBit*: Sets the bit at a specific position to 0.

- *AND* (`&`): Performs a bitwise "and" operation between two big-integers.

- *XOR* (`^`): Performs a bitwise "exclusive or" operation between two big-integers.

6.3.3 Basic Arithmetics

These operations are based on the classical algorithms as described by Donald Knuth in *The Art of Computer Programming* [35]. The implementation is inspired by TTMath [70] and available on the host and on the device implementation of our library.

- *addition* (+): Calculates the sum of two big-integer.

- *subtraction* (−): Calculates the difference between two big-integers.

- *multiplication* (⋆): Calculates the product of two big-integers.

- *division* (/): Calculate the integer quotient of two big-integers.

- *modulo* (%): Calculates the reminder of an integer division of two big-integer values.

6.3.4 Advanced Arithmetics

The following procedures are only implemented at the host code of the library. The squaring algorithm is a specialized multiplication which is used to speed up the exponentiation (*pow* and *modPow*).

- *square*: Specialized method that calculates the product of a big-integer times itself. It is faster than the general multiplication method, since it takes advantage of the fact that lots of primitive word multiplication results can be used twice (based on `lbnSquare_32` of Colin Plumb's C library [62], see also Section 2.2.3 "Fast Squaring" in Niall Emmart's Doctoral Dissertation [14]).

- *pow*: Calculates the big-integer whose value is a^x where a and x are both big-integers. It uses an iterative implementation of the *Exponentiation by Squaring* algorithms that was already stated in the Sanskrit book Chandah-Sûtra, about 200 BC (See section "2.2.9.1 Exponentiation by Squaring" of Niall Emmart's Doctoral Dissertation [14] for an explanation and Java `java.math.BigInteger` class [57] for an faster alternative.).

- *sqrt*: Computes the floored square root of a big-integer by using the *digit-by-digit* algorithm.

6.3.5   Number Representation Conversions

The following methods are used for creating big-integers with a specific value and also for printing them in human readable form. They are used for writing unit tests, for storing Paillier keys in text files and they are also useful for any kind of debugging. On the GPU no such conversion is required. Especially reading and writing string on a GPU would not make much sense. Therefore, these procedures are only implemented at the host code.

- *fromUint64*: Creates a big-integer instance that represents the same value as the provided C++ native 64 bit unsigned integer.

- *fromString*: Converts a number stored in a string as human readable characters to a big-integer representation. It supports numbers with a base between 2 and 16 (including). Therefore, the common bases 2 (binary), 10 (decimal), and 16 (hexadecimal) are supported.

- *toUint64*: Converts the lowest (least significant) 64 bit of a big-integer to a C++ native 64 bit unsigned integer.

- *toString*: Creates a string with human readable characters that represents the number stored in a big-integer. The procedure can convert into decimal (base 10) and hexadecimal (base 16) representations.

6.3.6   Special Algorithms for Asymmetric Cryptography

The procedures in this group are necessary for public key cryptography and only implemented at the host code. The *modPow* procedure will be required for an efficient Paillier multiplication with one encrypted and one plaintext value. Since the encrypted GPU renderer currently only supports X-ray rendering with nearest neighbor sampling, which means that the Paillier multiplication is not required, the *modPow* is not required at the GPU. However, all the more advanced rendering techniques explained in this thesis would require the *modPow* procedure on the GPU since they require the Paillier multiplication. The procedures *randomNumber* and *probablePrime* are used for Paillier key creation. *randomNumber* is furthermore used for the obfuscation during the encryption. The *modInverse* method is needed in many other algorithms like the Paillier decryption, subtraction of encrypted values, or the Montgomery multiplication [51] (at the pre-computable part).

- *gcd*: Calculates the greatest common divisor of two big-integers by the (standard) Euclidean algorithm [36].

- *modInverse*: Computes the multiplicative inverse of a big-integer in the residues class of another big-integer ($a^{-1} \mod n$, with the property $a^{-1} * a = 1 \mod n$). The heavy lifting is done by a highly optimized iterative implementation of the

extended Euclidean Algorithm [36] which uses shifting instead of divisions and works on unsigned integers. The implementation is based on the work of Laszlo Hars [23]. There exist already many implementations for various programming languages of the extended Euclidean algorithm. There are some implementations that can work on unsigned variables, there are iterative implementations, there are binary versions that use right shifting instead of expensive divisions, and there are also examples that have strict upper limits for the magnitude of intermediate results. However, a working example of an extended Euclidean (modular inverse) algorithm that brings together all these nice properties into a single algorithm is not publicly available. Therefore, we implemented the *iterative shifting unsigned extended Euclidean algorithm* that was adopted from Laszlo Hars [23] in Algorithm 6.1 and the final modular inverse algorithm that catches some special cases in Algorithm 6.2.

- *modPow*: A modular exponentiation ($a^x \mod n$) procedure that requires only about twice as many bits as a has for the storage of intermediate results. In order to achieve this, a reduction (modulo) is performed after each single multiplication which is much more efficient than a naive implementation that first calculates the exponentiation and then the modulo part. Further performance improvements were achieved through the use of the Montgomery representation [51] and the fixed window exponentiation approach (see section "2.2.9.2 Fixed Window Exponentiation" of Niall Emmart's Doctoral Dissertation [14]).

- *randomNumber*: Generates random big-integers with a definable bit length. ISAAC [29] is used as a fast cryptographic random number generator that generates 64 bit of random data per iteration. Therefore, the *randomNumber* procedure generates multiple random 64 bit words with ISAAC and copies them one after the other. Leading bits in the last word (most significant) will be masked out, if the requested bit length is not a multiple of 64.

- *probablePrime*: Returns true only if the input big-integer is a prime number with specified certainty. It uses the Miller-Rabin tests (see Appendix of NIST FIPS 186-4 [71]) and Lucas-Lehmer probable prime test [2] for testing and is based on the method `isProbablePrime` of Java `java.math.BigInteger` class [57]. This procedure is required for the Paillier private key creation where our implementation generates random numbers with half the bits of the requested public key length and tests this random number with the *probablePrime* procedure until it finds a number that is prime with a certainty of 100%.

---

[1]Unlike the normal extended Euclidean algorithm [36], this does not return the Bézout coefficients u and v, which would satisfy: $A \cdot u + M \cdot v = \gcd(A, M)$

Algorithm 6.1: Iterative shifting extended Euclidean algorithm with unsigned arithmetic. (Based on the function `SEUinv` from Appendix of L. Hars [23])

Parameters: The big-integer magnitude arrays A and M.

Result: The greatest common divisor (gcd) of A and M is returned as big-integer magnitude array. If A and M are coprime ($\gcd(A, M) = 1$) the output parameter A_{inv} provide the modular inverse $A^{-1} \mod M$. [1]

1   **procedure** `ISUExtEuclidean`(**in** A, **in** M, **out** A_{inv})
2   **if** $A < M$ **then**
3      $U = M$ $V = A$
4      $R = \text{BigInteger}(0)$ $S = \text{BigInteger}(1)$
5   **else**
6      $V = M$ $U = A$
7      $S = \text{BigInteger}(0)$ $R = \text{BigInteger}(1)$
8   **end**
9   **while** $V > \text{BigInteger}(1)$ **do**
10      $f = \text{bitLength}(U) - \text{bitLength}(V)$
11   **if** $U < (\text{shiftLeft } V \text{ by } f \text{ bits})$ **then**
12        $f = f - 1$
13   **end**
14      $U = U - (\text{shiftLeft } V \text{ by } f \text{ bits})$
15      $T = S$
16   **for** $i = 0$ **to** $f - 1$ **by** 1 **do**
17        $T = T + T$
18   **if** $T > M$ **then**
19          $T = T - M$
20   **end**
21   **end**
22   **while** $R < T$ **do**
23        $R = R + M$
24   **end**
25      $R = R - T$
26   **if** $U < V$ **then**
27        $\text{swap}(U, V)$ $\text{swap}(R, S)$
28   **end**
29   **end**
30   **if** $V = \text{BigInteger}(0)$ **then**
31   **return** U // A and M are not coprime $\implies A_{\text{inv}}$ does not exist.
32   **else**
33      $A_{\text{inv}} = S$ // A and M are coprime $\implies A_{\text{inv}}$ does exist.
34   **return** V // $= 1$
35   **end**

Algorithm 6.2: Compute the multiplicative inverse of a big-integer in a residues class.

Parameters: The big-integer magnitude arrays A and M.

Result: The modular inverse $A_{\mathrm{inv}} = A^{-1} \mod M$ is returned as big-integer magnitude array.

---

1 procedure `modInverse`(**in** A, **in** M)

2   **if** $M = $ `BigInteger(0)` **then**

3   Error: The Modulus M must be positive

4   **end**

5   **if** $M = $ `BigInteger(1)` **then**

6   **return** `BigInteger(0)`

7   **end**

8   GCD = `ISUExtEuclidean`(A, M, **out** A_{inv})   // For the classical Extended Euclidean algorithm [36] the multiplicative inverse A_{inv} could be calculated from the Bézout coefficients u of A by $A_{\mathrm{inv}} = u \mod M$.

9   **if** GCD $\neq$ `BigInteger(1)` **then**

10   Error: A does not have a multiplicative inverse in residues class M because the numbers are not relatively prime.

11   **end**

12   **return** A_{inv}

6.4 C++ Library

A big-integer library is a complicated peace of software where lots of details need to be done right in order to work correctly in every edge case. Therefore, it must be expected that many tests and debugging are necessary during the development. However, a GPU is a black box that is hard to debug. For this reason a prototype in C++ for the CPU was implemented before the actually work on the GLSL implementation for the GPU was started. For C++ on CPUs a number of sophisticated tools for testing and debugging exist, which can help to iron out problems such as algorithmic errors, overlooked edge cases, memory leaks and so on.

In C++ we started with an arbitrary length big-integer class that does not only contain methods for the operation required for encrypted X-ray rendering but also methods that can create instances from string or write the content of a big-integer to a string. Furthermore, methods required for creating keys for the Paillier cryptosystem (e.g. finding random prime numbers). All these methods benefit from the greater flexibility of integers that can grow very big on demand. However, big-integer representations that require dynamic memory allocation are not appropriate for a fast GPU implementation. Therefore, we developed a second big-integer class for fixed length integers based on the arbitrary big-integer class. This second big-integer class serves as a blue print for the GPU implementation in GLSL. The actual size is specified by a C++ class template

parameter. This allows the user of the library to support fixed big-integer with different lengths at the same application easily. Since GLS does not support templates like C++, the GLSL implementation uses a precompiled constant instead of a template parameter for specifying the actual big-integer size. In practice this is not a real limitation because the host program can compile the same shader with different big-integer size constant when required. Our library supports the conversion between fixed and arbitrary long big-integer, so it is possible to use public keys created by the arbitrary long integer class with values that are stored as fixed size integers. Since the goal here is to show a working GPU based homomorphic encrypted volume rendering we want to focus on the fixed big-integer class and the methods required for the actual rendering.

Apart from the big-integer classes the C++ host code also includes the Vulkan code that controls the GPU and some classes that encapsulates different aspects of the Paillier cryptosystem. The two most important Paillier classes are `PublicKey` and `SecureKey`. The class `SecureKey` contains the two big primes p and q, the decryption algorithm, and also a factory method for the creation of a new secure key. Furthermore, an instance of `SecureKey` holds a reference to the corresponding instance of `PublicKey`. The class `PublicKey` stores the modulus N and contains methods for encryption and the Paillier operations add and multiply.

6.4.1 Unit Tests

Automated unit tests were an integral part of the entire development work. The implementation of a big-integer operation did go hand in hand with the creation of unit test cases for it. Many times the unit test was implemented even before the actual library feature that should fulfill the test was implemented. For the project about 4,600 single assertions were written that test 107 use cases. The unit tests were not only useful during the initial implementation of operations, they were especially useful during performance optimizations of operations. Hence, it was possible to focus on the speed improvement first, without having to worry that the function will no longer work correctly for some edge cases without it being noticed.

The unit tests are written with the help of *catch2* [45]. This is a header only library for C++. It contains C++ pre-compiler macros for many common unit test writing tasks. Such as creating a test block with a name that encapsulates a sequence of assertions building on each other, and various types of assertions (require equality, require a special exception, and so on). *catch2* can also create a main function which compiles into a console application that executes the test, prints a comprehensive test summary and details about failed tests.

Unit tests are a very efficient tool for a big-integer library. At many systems that should be tested with unit test external dependencies such as database tables with special data in it are required. Such data needs to be created (mocking) before a test case can be executed which leads to a lot more time that is required for the test design and implementation. However, none of the arithmetic operations does have any external dependencies and,

therefore, does not need any mocking. Only some tests for the implementation of the Paillier cryptosystem required keys with specific sizes to be created upfront, but in fact even that is a really trivial mocking case. Therefore, unit tests are very suitable for testing a big-integer library.

6.5   GLSL Library

While the C++ part of the homomorphic-encrypted X-ray rendering on GPU project did consume most of the time, it was just the preparation for the big-integer library and finally the rendering on the GPU. The GLSL big-integer library is as far as possible a translation of the fixed length big-integer C++ class. Since GLSL lacks many features that C++ has, a simple translation is not always possible. The most relevant features that the C++ version of our library uses but GLSL does not support are classes, pointers and templates.

The absence of class constructs can be overcome by using a struct for the member variable (data part) of a class. Class member methods can be replaced by functions with name that contains the class name as prefix followed by the method name (e.g. the C++ method `add` in the class `UFixBigInt` will be the function `UFixBigInt_add` in GLSL). None static member method will require the instance data (e.g. `this->magnitude`) of the class, therefore, we establish the pattern that every GLSL function that is a translation of an C++ class member method must have a parameter called `me` as first parameter. Take the C++ class member methods `UFixBigInt UFixBigInt::add(const UFixBigInt & other) const` as an example. The meaning of this C++ method declaration is: a methods named `add` which is part of the class `UFixBigInt` that does return a object of type `UFixBigInt`, does require a reference (`&`) to a `UFixBigInt` as input parameter (`other`) which will not be changed (first `const`) and the method will not change anything on the member variables of the class instance for which it is called (last `const`). This C++ method will be translated in the following GLSL function declaration: `UFixBigInt UFixBigInt_add(const in UFixBigInt me, const in UFixBigInt other)`. If the C++ method is not `const` and, therefore, can write to the class member variables the first parameter of the GLSL function needs to be `inout UFixBigInt me`. Visibility modifiers like `public` and `private` are not possible in GLSL. However, we did clearly state the intended visibility into the doc-comment of every GLSL function (e.g. *@public*, *@private*). Therefore, a developer should be able to recognize which method should not be called directly even if the compiler does not enforce it.

GLSL does not support pointers, references or arrays with a dynamic length. Therefore, all pointers or references to big-integer magnitude storage arrays in C++ had to be replaced with fixed length arrays of unsigned integers. This is also one of the reasons why an arbitrary long integer library for Vulkan will be even much more challenging than a fixed length big-integer library. Since GLSL does not support passing parameters by reference all function parameters need to be passed by value. At least the declaration

needs to be written as passing by value. However, in order to help the GLSL / SPIR-V compiler to prevent memory copy operations as much as possible, we declared all method parameter as strict as possible. Therefore, a parameter that will not be changed inside a method is declared as `const`. If a parameter is only used for returning values, it is declared as `out`, therefore, the compiler never needs to create code that copies any value into the method. A parameter that only serves as input is declared as `in` so the compiler does not need to create code that copies any value out from the method. Only if a parameter really serves as input and output it is declared as `inout` which has the potential for the highest runtime complexity since the compiler could be forced to create code that copies e.g. a long array into the method and also out of the method.

The last missing language feature was the template. At the relevant C++ classes, the template feature was only used to specify the maximal word count of the big-integer magnitude storage array. However, this could be replaced by a pre-compiler constant in the GLSL implementation. Since we designed the C++ big-integer classes with the translation to GLSL already in mind, the translation process from C++ to GLSL went relatively smooth.

6.5.1 Big-Integers Stored in Vulkan Textures

Hence, we want to make use of the conventional graphics rendering pipeline of a GPU and furthermore want to use the texture units as smart voxel access buffers, the encrypted values which are represented as big-integers need to be stored in a texture format supported by Vulkan. This is accomplished by splitting one big-integer across multiple textures of the type `VK_FORMAT_R32G32B32A32_UINT`. This format holds four 32 bit unsigned integers, so 128 bits in total, per pixel or voxel. Therefore, a 1024 bits long big-integer will require 8 textures for storage. The C++ header file of the Vulkan SDK also defines the format `VK_FORMAT_R64G64B64A64_UINT` which could theoretically store 256 bits per pixel. However, non of the test platforms support a texture with 64 bit components as render target attachment.

The used splitting scheme is defined as following. The first (least significant) four 32 bit words of a big-integer are stored in the first texture. Thereby, the first word is stored in the first (red) channel, the second word in the second (green) channel, the third word in the third (blue) channel, and the fourth word is stored in the fourth (alpha) channel. The fifth, sixth, seventh, and eighth words of the big-integer are stored in the second texture. The fifth word in the red channel, the sixth in the green, the seventh into the blue and the eighth word is stored into the alpha channel of the second texture. The next four words of the big-integer are stored in the third texture and so on, until all words of the big-integer are stored in a channel of a texture.

When a shader needs to read a big-integer, the host will set all required textures as an uniform array to the shader. Then the shader code iterates over this array of textures and thereby it writes the values stored in the different channels into a local array of unsigned integers. So the shader basically reassembles the big-integer magnitude array

from the textures into which the host split the big-integer magnitude array. When a shader needs to write a big-integer it splits the big-integer magnitude array into multiple textures exactly like the host code. Texture arrays that contain split big-integers, written by a shader, can then be copied back from the device memory into the main memory where the host code can reassemble the big-integer magnitude arrays from the texture data. This is for example done after an encrypted image is rendered on the GPU, since all the encrypted pixel values that are split into multiple textures need to be written into big-integers so that they can be decrypted.

6.5.2   Automated Testing on the GPU

Tracing an error back from an encrypted image that cannot be decrypted successfully to the responsible lines of code would hardly ever be possible. Since there are more than 1000 lines of GLSL involved in rendering an encrypted image. Furthermore, an error could also lay in one of the memory copy operations or in one of the split into texture and reassemble into a big-integer magnitude array operation. Therefore, a fine-grained testing of the GPU code is required. Since we had great success with unit tests for the host code, we wanted something similar for the GPU code. For this reason we implemented a basic testing framework for the GPU part of the library.

The framework that we have developed for GPU code testing includes four C++ classes, a simple vertex shader, and a fragment shader. The C++ class `Assertion` encapsulates a list of big-integers which represent the parameters of a function that should be tested and the correct reference result which is also a big-integer. The class `BigIntTestCase` contains a list of assertions for one specific function that should be tested. Beside the assertions it also contains a name that is used for the console output and the operation type which basically specifies the method that should be tested. These two classes serve only as a structured storage and do not contain any notable logic. The `BigIntTestFactory` class is basically the place where all assertions are stated and grouped into `BigIntTestCase`.

All of the interesting processing is done in the class `BigIntTestObj`. For every `BigIntTestCase` a texture is created. If the big-integer bit length of the test case does exceed 128 bit, multiple textures will be created (see last section). The height of the textures is equal to the count of `Assertions` in the `BigIntTestCase` and the width is equal to the parameter count of the function that should be tested. After the textures are allocated, all parameters of all assertions in the `BigIntTestCase` will be copied to these textures. The parameters of the first assertion will be stored in the first row of the textures, the parameters of the second assertion go into the second row and so on. A second set of texture will also be created. It is used as storage for the result from the GPU operations. These textures have the same height as the textures of the first texture set, thus the textures have as many rows as the test case has assertions. The width of the textures is always one. This second set of textures will be used as color attachments for the target frame buffer to which the test fragment shader will write to. A Vulkan rendering pipeline with this frame buffer is set up. The vertex shader of

this pipeline always draws a screen filling quad (two triangles), which has a pixel size of one times the assertion count. Before the draw call is submitted the host code needs to configure the fragment shader so that it executes the operation (function) that the test case requests for. For that purpose we did use a uniform integer variable that is used as the parameter of a long switch statement in the fragment-shader. The cases of the switch statement then contain the different function we want to test. While this trivial "configuration" approach works well for us, it probably does not scale well. Therefore, we want to suggest to create the fragment shader code dynamically before setting up the rendering pipeline. A template for the shader code could contain just a placeholder for the actual function call which will be replaced for each test case. The actual function call for a specific test case could be added to the `BigIntTestCase` class. While the draw call is running on the GPU the fragment shader is executed for each row exactly once. Therefore, for every assertion the fragment shader is executed exactly once. The fragment shader grabs the function input parameters from one raw of the input textures. The row index for the texture lookup is equal to the row index of the fragment that the shader is meant to draw. After re-assembling the big-integer magnitude arrays of the function parameters from the words stored in the different texture pixels, the shader calls the function that should be tested. The result of the function call is then split into 128 bit parts and written to the textures of the target frame buffer. The host code waits until the GPU has finished the draw call, which means it waits until all fragments are drawn. When the GPU is done, the host code reads back the results from the textures used as frame buffer attachments and reassembles the big-integers from the pixel. These big-integers now contain the results for each assertion and will be compared with the reference results. If the big-integer from the GPU and the reference result stored in the `Assertion` object are not equal an error will be printed. If the two big-integers are equal only a success counter is incremented. This counter is used to print a final summary after all test cases are executed.

Some of the tested methods do not return a big-integer, e.g. the comparison operators which only returns true or false. In such a case we encoded the reference result and the result from the GPU function into a big-integer. For the comparison operators, which returns a boolean result, the least significant bit can be used to store true (1) or false (0). If all other bits of the big-integer are set to zero, the comparison between the GPU result and the reference result has still the desired effect and a failed test will be detected correctly.

6.5.3 Big-Integer Length Limits

While using the standard Vulkan rendering pipeline is beneficial for a proof of concept implementation, it has a significant disadvantage for a production setup, since the maximal count of bits a fragment shader can write is quite limited on current hardware. We did not have access to any GPU that supports more than 8 fragment shader output attachment (see `maxColorAttachments` and `maxFragmentOutputAttachments` of `VkPhysicalDeviceLimits` [74]), nor could we find any GPU that supports more

in the Vulkan Hardware Database [80]. Since a fragment shader can only write 32 bits per channel and there is no texture with more than 4 channels, the total count of bits that can be written by a fragment shader is $8 \cdot 32 \cdot 4 = 1,024$ bits.

A Paillier encrypted number has twice as many bits as the modulus of the public key. Therefore, the approach with the standard rendering pipeline is limited to $1,024/2 = 512$ bit long public keys, which cannot be considered as secure (see Section 8.2). For an implementation that requires longer public keys compute shaders can be used instead of fragment shaders. Hence, compute shaders do not have such limitation and can write as many bits to the device memory as required. The GLSL library for big-integer arithmetic that we implemented could also be used for a Vulkan compute shader. However, for compute shader the voxel access with the help of 3D texture units needs to be replaced by a slightly different GPU implementation.

The bit count of the modulus of the public key times two is only the storage size. Let l_N be the length of the public key modulus in bits. So the required storage size is $2l_N$. However, the size required for the actual calculation will be larger. The exact size depends on the operations that should be performed and on the used algorithms. The X-ray rendering with nearest-neighbor sampling requires only the big-integer arithmetic operations multiply and modulo. The size of a multiplication result is equal to the sum of the sizes of the factors. Therefore, a multiplication of two Paillier encrypted values with a size of $2l_N$ will lead to a $4l_N$ bit long integer. The school multiplication algorithm (shown in Algorithm 6.7 and Algorithm 6.8) does not require any temporary big-integer that is larger than the final result. The modulo operation does not need any big-integer variable other than the value that should be reduced, therefore, it does not require longer integers than the integers to which it is applied. Since the modulo operation that reduced the multiplication result from a $4l_N$ bit long integer to a $2l_N$ bit long integer can be performed after every single multiplication the longest big-integer that is required for X-ray rendering with nearest-neighbor sampling is $4l_N$ bit long. However, the Montgomery multiplication [51] will require slightly longer big-integers for the calculation but not for the volume or result storage. See Section 6.5.4 for further details about the Montgomery multiplication and a discussion about the big-integer length requirements of it.

In this section we will discuss some code improvements we made that lead to the different runtimes shown in Table 7.4 at the results (Chapter 7). Our first functional implementation of encrypted X-ray rendering on GPU was based on the basic school multiplication stated in Algorithm 6.7 and the division algorithm from Knuth [35] (see also TTMath [70]). These operations are used to implement the Paillier add operation ($\oplus$). Which is used to sum up all the samples along a viewing ray. The two encrypted values that should be summed up in the plaintext domain need to be multiplied in the encrypted domain. In order to keep the magnitude of the sum as small as possible the value is reduced after each multiplication by calculation $\mod N^2$, for which the division is required. However, the performance was not satisfying and far from the speedup we were hoping for. For example, with a 512 bit public key an Nvidia RTX 3090 was only 5.6 times faster than a single core of a mobile Intel i7 with java (see Table 7.4). For

smaller keys the speedup was better but also not impressive. E.g. 67 times for a 128 bit key and 19 times for a 256 bit key.

The division algorithm has more lines of code than any other single arithmetic operation we implemented and is also the most complicated operation of the whole encrypted X-ray shader. The quotient digit estimation and correction of the division algorithm requires some if and loop statements where the condition depends on the input arguments. Conditions that are dependent on data, that is different for each thread, should be avoided on GPUs since it will lead to branching. Therefore, it can be assumed that the division has the greatest potential for savings.

6.5.4   Montgomery Multiplication

Since, the division result is actually not required and only the remainder is important, the costs that are associated with the normal division can be avoided by an approach that can calculate modulo N^2 without a normal division. The Montgomery reduction [51] achieves exactly that by transforming the numbers that should be multiplied into a special representation called Montgomery form, where the modulus N becomes R (we will call R also the *reducer*) and the only division required for calculating the modular reduction is the division by R. The reducer R must be co-prime to N and greater than N. The important thing now is that R can be chosen in such a way that it is a power of two (2^{l_R} - see line 3 of Algorithm 6.6). This is important, because a division with a number that is a power of two can be replaced by a shift right (see line 8 of Algorithm 6.6 - operator: `shiftLeft` X by l_R bits), and a modulo can be replaced by a bitwise and (see line 7 of Algorithm 6.6 - operator: **&**). The shift right (`shiftRight` X by l_R bits) and the bitwise and (**&**) operations are usually very fast instructions on processors.

With the Montgomery representation it is not only possible to perform multiplication in the residues class N by the common algorithm (see Algorithm 6.5) but also addition, subtraction, and some other operations. In the remainder of this section we will discuss the aspects of the Montgomery multiplication that are relevant for the homomorphic encrypted X-ray rendering. For further detail about Montgomery multiplication itself I refer the interested reader to the paper from Peter Montgomery [51].

The conversions to Montgomery form (see Algorithm 6.3) and from Montgomery form to the normal number representation (see Algorithm 6.4) are expensive operations, that have a higher runtime complexity than the normal division, therefore, it does not make sense to use a Montgomery reduction if only a single multiplication and reduction should be performed. However, if many multiplications and reductions need to be performed, the conversions do pay off. That's why, it is often used for modular exponentiation (*modPow*) where only the base value needs to be converted into Montgomery form and all further computation is done on this converted value or a value that is derived from it by multiplications and reductions until the exponentiation algorithm has finished and the final result is converted out of Montgomery form. (Examples for modular exponentiations that uses Montgomery reductions: BigNum from Colin Plumb [62], Java

`java.math.BigInteger` class [57], GPUMP from Kaiyong Zhao and Xiaowen Chu [82], Andrew Moss, Daniel Page and Nigel P. Smart [53], Owen Harrison and John Waldron [22].)

The sample composition for Paillier encrypted X-ray rendering also performs many multiplications and reductions in a row. However, the sample composition multiplies at every sample position a new value with the current accumulation buffer. That means that for every multiplication one of the two factors is a totally new value that needs to be converted into Montgomery form first. Therefore, the rendering of one encrypted image will require as many "to Montgomery form" conversions as it requires multiplications. This approach would lead to more overhead induced by the conversions than it can save on divisions or rather reduction, hence it is pointless. However, it can be assumed that usually not only one image will be rendered from a volume dataset but many images. Furthermore, the conversion into Montgomery form does not need to be part of the rendering, it can be part of the volume encryption. Consequently, there is no overhead for the Montgomery reduction during the ray traversal, but the saving in runtime complexity for the easier reduction after each multiplication remains. So, the reason why Montgomery multiplication is efficient for the X-ray rendering is a little different from the reason that makes Montgomery multiplication efficient for modular exponentiation, but nevertheless it is absolutely advisable to use the Montgomery multiplication for the Paillier encrypted X-ray rendering. The conversion out of the Montgomery form needs to be performed for every pixel of the final image. However, the saving should well pay off for the overhead of the conversion.

The Montgomery implementation in our big-integer library is very close to the algorithms shown in this section and the runtime measurement for the GPU results in Chapter 7 are done with this implementation. However, there are various opportunities for faster implementations. A review of Colin Plumb's C library [62] and Java `java.math.BigInteger` class [57] should be a good starting point for optimizations of the shown Montgomery reduction (Algorithm 6.6), conversion to Montgomery form (Algorithm 6.3), and converting back to the normal form (Algorithm 6.4).

In the context of the Montgomery multiplications the letter N is commonly used for the modulus that defines the residues class. Therefore, we also used it in this section. However, this variable N is not the modulus of the Paillier public key, it is just the residues class for which the Montgomery reduction should by applied. If the Montgomery reduction is used instead of the standard modulo operation for the Paillier algorithms `encrypt` (Algorithm 3.2), `add` ($\oplus$, Equation 3.1), or `multiple` ($\otimes$, Equation 3.2) the modulus N of the Montgomery algorithms will contain the squared modulus of the public key (which is stated as N^2 in the other sections of this thesis).

Required Big-Integer Size for Montgomery Multiplication

In this section we will analyze the maximal required size of a big-integer magnitude storage for the Montgomery multiplication in dependent of the Paillier public key modulus length

l_N. We will start with the pre-computable values l_R, R, M, R_{inv} and N' followed by procedures `reduce` (Algorithm 6.6), `toMont` Algorithm 6.3, `fromMont` (Algorithm 6.4), and `multiply` Algorithm 6.5.

The reducer bit count l_R will have 8 bits more than N in the worst case (see line 2 of Algorithm 6.6). In line 3 of Algorithm 6.6 a big-integer containing 1 is shifted by l_R bit to the left and stored in R, this leads to a worst case size of $2l_N + 9$. The bit mask M that is calculated in line 4 has exactly one bit less than R, since the binary representation of R has a leading one followed by only zeros, consequently a subtraction of one will turn the leading one into a zero and flips all following zeros to ones $(100\ldots000 \rightarrow 011\ldots111)$. The multiplicative inverse R_{inv} of R, which is calculated in line number 5 of Algorithm 6.6, can not be larger than N consequently R_{inv} does have a worst case size of $2l_N$. In line 6 the product of R with a size of $2l_N + 9$ bits and R_{inv} which has a size of $2l_N$ bits is calculated. So the product of R and R_{inv} has a worst case length of $4n + 9$. After the subtraction of 1 the number will be divided by N ($2l_N$ bits) which decreases the length by $2l_N$ bits. Therefore, N' will have a length of $(4l_N + 9) - 2l_N = 2l_N + 9$ bits.

The input parameter T of the procedure `reduce` can have a bit length of $4l_N$, since it contains the product of two big-integers in Montgomery form (see line 2 of Algorithm 6.5). In line 7 the bits of T that are above the $2l_N + 8$ least significant bits are masked out $(T \ \& \ M)$, hence the result has a maximal length of $2l_N + 8$ bit. This result is multiplied by N' with $2l_N + 9$ bit which leads to a product with $(2l_N + 8) + (2n + 9) = 4l_N + 17$ bits. The bits above $2l_N + 8$ of this product are again masked out $((\ldots) \ \& \ M)$ consequently the variable T_m requires maximal $2l_N + 8$ bits. The first operation that is executed in line 8 is the multiplication of T_m and N which leads to a $(2l_N + 8) + 2l_N = 4l_N + 8$ bits long integer. Then T with a maximal size of $4l_N$ bit is added. That leads to an intermediate result with a maximal length of $\max(4l_N, 4l_N + 8) + 1 = 4l_N + 9$ bits. The shift right (line 8) and the subtraction can only decrease the bit length, therefore, it is not relevant for the maximal required big-integer length. The final result that is returned by the procedure must be less than N and, therefore, can have maximal $2l_N$ bits (see line 9 to 13). Therefore, we can conclude that the maximal big-integer size that is required during the execution of the procedure `reduce` is $4l_N + 17$ bit, which can occur as intermediate result in line 7.

For the procedure `toMont` (Algorithm 6.3) we can safely assume that the input parameter P is a Paillier encrypted value less than the public key modulus and, therefore, has a maximal length of $2l_N$ bits. In line 3 P is shifted $2l_N + 8$ bits to the left at the worst case. This leads to an intermediate result of $2l_N + (2l_N + 8) = 4l_N + 8$ after the shift operation. The length of the shift operation result is then brought down to $2l_N$ bits by the final modulo operation. So the longest integer that occurs at the procedure `toMont` has $4l_N + 8$ bits.

The lines 2 to 4 of the procedure `fromMont` (Algorithm 6.4) define the pre-computable values l_R, R, and R_{inv} which are already discussed with the procedure `reduce`. In line 5 the product of E and R_{inv} is calculated. E is a number in Montgomery form and we can safely assume that it is less than N, since it is either a result of the conversion to

Montgomery form (see Algorithm 6.3) or a result of the Montgomery reduction (see Algorithm 6.6) and both procedures return only values less than N. Consequently, E cannot be longer than $2l_N$ bits. The multiplicative inverse of the reducer (R_{inv}) also has a maximal size of $2l_N$ bits (see above). Therefore, the product of E and R_{inv} has a maximal length of $2l_N + 2l_N = 4l_N$ bits. The following modulo operation will reduce the size back to $2l_N$ bits. So the longest integer that occurs at the procedure `fromMont` has $4l_N$ bits.

The last Montgomery related procedure is the multiplication stated in Algorithm 6.5. For the input parameters A and B we can safely assume that they are less than N and, therefore, have a maximal size of $2l_N$ bits, because, they need to be big-integers in Montgomery form returned by the procedure `toMont` (Algorithm 6.3) or `reduce` (Algorithm 6.6). In line 2 the big-integers A and B are multiplied, this leads to a $2l_N + 2l_N = 4l_N$ bits long number, which is reduced in line 3 by the Montgomery reduction (Algorithm 6.6) to a $2l_N$ bits long integer.

The worst case length of any temporary big-integer variable in the discussion about the four Montgomery procedures is $4l_N + 17$ bits at the line number 7 of Algorithm 6.6. For that reason we can conclude that the maximal bit length that needs to be supported in order to be able to use Montgomery multiplication for Paillier encrypted values, is four times the public key modulus length plus additional 17 bits ($4l_N + 17$).

Algorithm 6.3: Conversion of a normal (plain) big-integer to Montgomery form. (Based on the Montgomery reduction Java code from Project Nayuki [56])

Parameters: The "plain" big-integer magnitude array P which should be converted to Montgomery. The global big-integer N is the modulus.
Result: The big-integer magnitude array E which contains P in Montgomery form.

1 procedure `toMont` (**in** P)

2 $\quad l_R = \left(\lfloor \frac{\texttt{bitLength}(N)}{8} \rfloor + 1 \right) \cdot 8$ $\qquad\qquad$ } can be pre-computed for N

3 $\quad E =$ `shiftLeft` P by l_R bits) $\mod N$

4 $\quad$ **return** E

Algorithm 6.4: Conversion of a big-integer in Montgomery form to its normal (plain) representation. (Based on the Montgomery reduction Java code from Project Nayuki [56])

Parameters: The big-integer magnitude array E in Montgomery form which should be converted to its normal (plain) representation. The global big-integer N is the modulus.

Result: The big-integer magnitude array P which contains E in normal (plain) form.

1 **procedure** fromMont(**in** E)

2 $\quad \Big\vert \quad l_R = \left(\lfloor \frac{\text{bitLength}(N)}{8} \rfloor + 1 \right) \cdot 8$ $\qquad$ $\left. \begin{array}{l} \\ \\ \end{array} \right\}$ can be pre

3 $\quad \Big\vert \quad R = \text{shiftLeft BigIntegr}(1) \text{ by } l_R \text{ bits}$ $\qquad$ computed for N

4 $\quad \Big\vert \quad R_{\text{inv}} = R^{-1} \mod N \quad$ // multiplicative inverse
$\qquad\qquad\qquad\qquad\qquad$ of R in residue class N

5 $\quad \Big\vert \quad P = (E \cdot R_{\text{inv}}) \mod N$

6 $\quad \Big\vert \quad$ **return** P

Algorithm 6.5: Multiplication of two big-integers in Montgomery form modulo another big-integer. (Based on the Montgomery reduction Java code from Project Nayuki [56])

Parameters: The big-integer magnitude arrays A and B in Montgomery form which should be multiplied in the residues class N (global big-integer).

Result: The Product of A and B as big-integer magnitude array P in Montgomery form. ($P = \text{toMont}(\text{fromMont}(A) \cdot \text{fromMont}(B) \mod N)$).

1 **procedure** multiply(**in** A, **in** B)

2 $\quad \Big\vert \quad P_t = A \cdot B$

3 $\quad \Big\vert \quad P = \text{reduce}(P_t)$

4 $\quad \Big\vert \quad$ **return** P

Algorithm 6.6: Reduction of a big-integer in Montgomery form modulo another big-integer. (Based on the Montgomery reduction Java code from Project Nayuki [56])

Parameters: The big-integer magnitude array T in Montgomery form which should be reduced modulo the global big-integer N.

Result: The big-integer magnitude array T_t which contains the Montgomery form of "`fromMont`(T) mod N".

1 procedure `reduce`(**in** T)

2 $l_R = \left(\lfloor \frac{\text{bitLength}(N)}{8} \rfloor + 1\right) \cdot 8$

3 $R = $ `shiftLeft BigInteger(1)` by l_R bits

4 $M = R-$ `BigInteger(1)`   // the bit mask   can be pre computed for N

5 $R_{\text{inv}} = R^{-1} \mod N$ // multiplicative inverse of R in residue class N

6 $N' = \frac{R \cdot R_{\text{inv}} - \text{BigInteger}(1)}{N}$

7 $T_m = ((T \ \& \ M) \cdot N') \ \& \ M$ // Compare with "$m = (T \mod R) \cdot N' \mod R$" from Montgomeries `REDC` function [51]

8 $T_t = $ `shiftRight` $(T + T_m \cdot N)$ by l_R `bits`   // Compare with "$t = (T + m \cdot N)/R$" from Montgomeries `REDC` function [51]

9   **if** $T_t \geq N$ **then**

10 $T_t = T_t - N$

11   **end**

12   assert $T_t < N$

13   **return** T_t

6.5.5 School Multiplication Optimization

The first rendering runtime tests with the Montgomery multiplication have shown only very little improvements over the conventional approach that requires a standard division for the modulo reduction. Results for the modular exponentiation *modPow* (see Section 6.3.6) of the Paillier encryption on the CPU were even slower than the original implementation without Montgomery multiplication. Also the rendering on the AMD RX 580 becomes slower for 256 bit and 512 bit public keys (compare the "mul school" lines with and without Montg. in Table 7.4).

It can be seen from Algorithm 6.5 and Algorithm 6.6 that the Montgomery multiplication requires one shift right (Algorithm 6.6 line 8), one addition (Algorithm 6.6 line 8), two bitwise and (Algorithm 6.6 line 7), three multiplications (Algorithm 6.5 line 2 and Algorithm 6.6 line 7), one comparison (Algorithm 6.6 line 9) and sometimes one

subtraction (Algorithm 6.6 line 10). Let s be the count of words of a big-integer. The operations addition, subtraction, bitwise and, shift right, and comparison has a runtime complexity of $\Theta(s)$. While the multiplication has a complexity of $\Theta(s^2)$, since every word of the first big-integer needs to be multiplied with every word of the second big-integer (see the for-loops in Algorithm 6.7, Algorithm 6.8 and Algorithm 6.10). So the multiplication is not only the most complex operation but also becomes the most used operation at the Montgomery multiplication and, hence, also the most used operation for the total encrypted X-ray rendering. Consequently, the next optimizations address the multiplication.

The basic school multiplication algorithm stated in Algorithm 6.7 was the first implementation and, therefore, the reference. The line 3 to 6 determined the indices of the least significant words that are not zero (b_A, b_B) and the word counts without leading zeros (s_A, s_B), in oder to prevent unnecessary multiplications with words that are zero. Furthermore, the word length without leading zeros (s_A, s_B) is used in line 7 for an early overflow detection. In line 10 the big-integer R, that stores the result, is initialized and all words of that new big-integer will be set to 0. The for-loops in line 12 and 13 iterate over all non zero words of the input operands A and B. The procedure `mulTwoWords` (see Algorithm A.5) multiplies one 32 bit word from A and one word 32 bit from B and returns the 64 bit result in two 32 bit words, the lower (least significant) word r' and the high (most significant) word r''. The procedure call `addTwoInts` (see Algorithm A.3) in line 15 adds the two word multiplication result (r', r'') to the result R. The lower word r' will be added to the word at the index $i_B + i_A$ of R. The carry of this addition and the high word r'' will be added to the word at the next index ($i_B + i_A + 1$) of R. The procedure `addTwoInts` will only perform two single word add operations in best case, so the best case runtime is constant and independent from the total word count of the integer. However, the carry propagation can lead to n add operations if every word at the big-integer R has already reached its maximum value and the carry needs to be propagated through all words of the big-integer above the index $i_B + i_A + 1$.

If the result is so large that it will use the most significant word that is available in R, some special handling for the last multiplication is required. Therefore, the last multiplication is not done inside the loop (see "**while** $i_A + i_B < s_{\max} - 1$" in line 12) but in the if block from line 21 to 30. After the multiplication in line 22, the if in line 23 checks, if the higher 32 bit word of the result (r'') is greater than 0 which would lead to an overflow. After adding the lower word of the multiplication result (r') to R in line 26, the if in line 27 checks if this addition returns a carry flag. If a carry flag is returned, the multiplication is not possible without an overflow. All the overflow handling is of course only required for the fixed size big-integers, but not for the arbitrary long big-integers.

Deferred Carry Propagation

In order to enhance the runtime for the multiplication we adopted the big-integer multiplication algorithm from Colin Plumb's C library [62]. The Algorithm 6.8 shows the pseudocode of the version we created for fixed size big-integer. Which is also a bit

more readable than the original from Colin Plumb [62] or Java version [57]. It also represents a school multiplication with one adjustment. The carry that can occur when the multiplication result of two words is added to R is not propagated immediately (as it is done at the Algorithm 6.7) but stored and added to the result of the next word multiplication (see the lines 6 and 8 of Algorithm 6.9, and the lines 13 and 15 of Algorithm 6.8).

In line 3 and 4 of the Algorithm 6.8 the word count of A and B without leading zeros is determined in order to prevent unnecessary multiplications with zero. If the word count of A (s_A) or B (s_B) is only one (s_A and s_B are also one if A or B is zero - see `wordLength` in Section 6.2) a simplified multiplication algorithm called `mulInt` (see Algorithm A.6), that can only multiply a single word with a big-integer, will be used (stated in lines from 6 to 9). The line 5 creates a new big-integer variable R that will hold the result and set all words of it to zero. The loop from line 11 to 19 iterates over all words of the input factor A. The second loop that is logically required for a multiplication of two multiple words long numbers is part of the procedure `mulAdd` (Algorithm 6.9) called in 13.

At the call of `mulAdd` the result R is provided as the first parameter (name A) which is used as input and output. The second parameter i_A specifies the index of the first parameter A where the first multiplication result will be stored. The third parameter (B) is the big-integer whose words should be multiplied with the current word $A[i_A]$ which is provided as last parameter named k. The next parameter is named i_B and a hardcoded 0 is provided for it. It defines the index of the previous parameter B where the multiplication starts. As second last parameter (named n) the size of B (s_B) is provided. The parameter n specifies how many words from parameter B, starting at index i_B, should be multiplied with the last parameter which is named k. Note that the big-integer provided as first parameter (named A) needs to be long enough to hold at least $i_A + n$ words.

The procedure `mulAdd` (Algorithm 6.9) requires machine words that are twice as long as the common words used for the big-integer magnitude storage (or at least the compiler support for it). Since we use 32 bit words as storage for our test, this procedure requires support for 64 bit words and arithmetics. This is for sure no problem on 64 bit CPU and also was no problem for the SPIR-V compiler and the tested graphics cards. The variables which need to be double length are recognizable by the subscript $_{\text{long}}$. A cast from a normal word length variable X to a double word length variable X_{long} is represented by $X_{\text{long}} = \text{long}(X)$. The opposite cast from a double word length variable X_{long} to a normal word length variable X is stated as $X = \text{short}(X_{\text{long}})$. It drops the upper 32 bit and stores only the lower 32 bit to the smaller variable.

In line 3 of Algorithm 6.9 the multiplication factor k is cast into a double width variable k_{long}. Then the double width variable c_{long}, that will temporarily store the carry which needs to be propagated to the next multiplication, is initialized with zero. The remaining part of the procedure is the loop that is repeated n times. The line 6 performs the actual word multiplications ($\text{long}(B[i_B]) \cdot k_{\text{long}}$), the addition of the result from a

previous call ($+\texttt{long}(A[i_A])$), and the deferred carry propagation ($+c_{\text{long}}$). All that computations are done with 64 bit arithmetics and the result is stored in c_{long}. In line 7 the lower 32 bit of c_{long} are stored to the output (and input) parameter A at index i_A. The shift right operation in line 8 replaces the 32 least significant bits of c_{long} with the 32 most significant bits of the same variable. The higher 32 bits will be set to 0 by the shift right. Now the c_{long} contains the carry which is deferred for the next iteration. The code in lines 9 and 10 increment the indices i_A and i_B for the big-integers A and B. Finally, the last line returns the last carry which will be stored in R by the calling procedure $\texttt{multiplyDeferredCarry}$ (line 15 of Algorithm 6.8) and so it will be propagated to the next call of $\texttt{mulAdd}$, where it will be used as value of $\texttt{long}(A[i_A])$ in line 6 during the last iteration of the loop. If it is the last call of $\texttt{mulAdd}$ from $\texttt{multiplyDeferredCarry}$ it can also lead to an overflow (line 17 of Algorithm 6.8).

The runtime complexity of this multiplication (Algorithm 6.8 and Algorithm 6.9) is $\Theta(n^2)$. Therefore, it has the same theoretical complexity as the first multiplication Algorithm 6.7. However, in practice this is faster as it can be seen from the runtime results in Table 7.4.

Reduced Branching

Branching on GPUs comes at high cost in terms of runtime since it prevents the parallel execution of threads that need to go through different code blocks. Therefore, we tested a version of the deferred carry propagation multiplication (Algorithm 6.8), where we reduced all possible branching as much as possible. The resulting version that should lead to reduced branching is apparent in Algorithm 6.10. It does not have the special treatment for multiplications where at least one of the two factors does have zero or only one word. So, the lines 5 to 9 of Algorithm 6.8 are missing in Algorithm 6.10. The search for the word count without leading zeros in line 3 and 4 of Algorithm 6.8) are also replaced in Algorithm 6.10, where s_A and s_B are just set to the maximal word count of the fix size big-integer (s_{max}). This has the disadvantage that the procedure $\texttt{mulAdd}$ (Algorithm 6.9) will execute many single word multiplications that are not necessary since one of the factors will be 0 and, therefore, the result will be 0 anyway. However, it should create fewer branches. Another condition that can lead to branching is the overflow detection in line 16 and 17 of Algorithm 6.8 which is also not present in Algorithm 6.10.

To make it short, the reduced branching does not pay off for the unnecessary multiplications with words that are 0. This was expected for the volume encryption with Paillier on the CPU where the modular exponentiation *modPow*, which heavily uses the multiplication, is responsible for the overwhelming part of the total runtime. However, even the rendering on the GPU cannot benefit enough from the reduced branching in order to mitigate the runtime of the unnecessary multiplications. This can be observed by comparing the line for the deferred carry propagation multiplication (mul optimized) with the lines for the reduced branching multiplication (mul opt. & red. bran.) in Table 7.4).

Algorithm 6.7: Big-integer school multiplication. (Based on TTMath [70])

Parameters: The big-integer input factors A and B.
Result: The big-integer product R of A and B

1 procedure multiplySchool(**in** A, **in** B)

2 $s_{\max}$ = the maximal word count of a BigInteger

3 s_A = wordLength(A)   // size of A in words

4 b_A = findLowestSetWord(A)   // begin of A in words; can be hardcoded to 0

5 s_B = wordLength(B)   // size of B in words

6 b_B = findLowestSetWord(B)   // begin of B in words; can be hardcoded to 0

7   **if** $s_A + s_B - 1 > s_{\max}$ **then**

8   Error: Multiplication not possible without overflow.

9   **end**

10 R = new BigInteger with all words set to 0.

11 $c = 0$

12   **for** $i_A = b_A$ **to** $s_A - 1$ **by** 1 **do**

13   **for** $i_B = b_B$ **to** $s_B - 1$ **by** 1 **while** $i_A + i_B < s_{\max} - 1$ **do**

14   mulTwoWords($A[i_A]$, $B[i_B]$, **out** r'', **out** r')

15 $c = c +$ addTwoInts(r'', r', $i_B + i_A$, **inout** R, $s_{\max}$);

16   **end**

17   **end**

18   **if** $c > 0$ **then**

19   Error: Multiplication not possible without overflow.

20   **end**

21   **if** $s_A + s_B - 1 = s_{\max}$ **then**   // multiply with last word if required

22   mulTwoWords($A[s_A - 1]$, $B[s_B]$, **out** r'', **out** r')

23   **if** $r'' > 0$ **then**

24   Error: Multiplication not possible without overflow.

25   **end**

26 $c =$ addInt(r', $s_{\max} - 1$, **inout** R, $s_{\max}$)

27   **if** $c > 0$ **then**

28   Error: Multiplication not possible without overflow.

29   **end**

30   **end**

31   **return** R

Algorithm 6.8: Optimized big-integer school multiplication with deferred carry propagation. (Based on Colin Plumb's BigNum math library [62])

Parameters: The big-integer input factors A and B.
Result: The big-integer product R of A and B

1 procedure multiplyDeferredCarry(**in** A, **in** B)
2 $s_{\max}$ = the maximal word count of a BigInteger
3 $s_A = $ wordLength(A) // size of A in words
4 $s_B = $ wordLength(B) // size of B in words
5   **if** $s_B = 1$ **then**
6   **return** mulInt$(A, B[0])$
7   **else if** $s_A = 1$ **then**
8   **return** mulInt$(B, A[0])$
9   **end**
10 $R = $ new BigInteger with all words set to 0.
11   **for** $i_A = 0$ **to** $s_A - 1$ **by** 1 **do**
12   assert: $s_B - 1 + i_A < s_{\max}$
13 $c = $ mulAdd(**inout** R, i_A, B, *0*, s_B, $A[i_A]$)
14   **if** $s_B + i_A < s_{\max}$ **then**
15 $R[s_B + i_A] = c$
16   **else if** $c > 0$ **then**
17   Error: Multiplication not possible without overflow.
18   **end**
19   **end**
20   **return** R

Algorithm 6.9: Multiplication of a big-integer magnitude array by a single word integer and a further addition of the product to the destination big-integer magnitude array. (Based on Colin Plumb's BigNum math library [62])

Parameters: Multiplies n words from the the big-integer magnitude array B starting at index i_B with the single word integer k and add the products to the big-integer magnitude array A starting at index i_A. This function requires the support of integers that are twice as long as the integers used as words for the big-integer magnitude arrays (e.g. 64bit unsigned inters are required if the big-integer magnitude arrays used 32 bit unsigned integers.) The variables that require twice as much bits as the common words type are marked by the subscript x_{long}. The conversion (cast) of a word with the common length to a word with the double length is expressed as $x_{\text{long}} = \texttt{long}(x)$.

Result: The result of the multiplication is added to A and provided by the first parameter. A potential single word carry value is provided by the return value.

---

1   **procedure** $\texttt{mulAdd}$ (**inout** A, **in** i_A, **in** B, **in** i_B, **in** n, **in** k)
2   asset $n > 0$
3     $k_{\text{long}} = \texttt{long}(k)$
4     $c_{\text{long}} = 0$
5   **for** $i = 0$ **to** $n - 1$ **by** 1 **do**
6       $c_{\text{long}} = \texttt{long}(B[i_B]) \cdot k_{\text{long}} + \texttt{long}(A[i_A]) + c_{\text{long}}$
7       $A[i_A] = \texttt{short}(c_{\text{long}})$
8       $c_{\text{long}} = \texttt{shiftRight}\ c_{\text{long}}$ by the bits of the common word (e.g. 32bit)
9       $i_A = i_A + 1$
10      $i_B = i_B + 1$
11   **end**
12   **return** c_{long}

Algorithm 6.10: Optimized big-integer school multiplication with reduced branching. (Based on Colin Plumb's BigNum math library [62])

Parameters : The big-integer input factors A and B.
Result: The big-integer product R of A and B

1 **procedure** `multiplyReducedBranching`(**in** A, **in** B)
2　　$s_{\max}$ = the maximal word count of a BigInteger
3　　$s_A = s_{\max}$ 　　　　　　　　　　　　　　　　　　　// size of A in words
4　　$s_B = s_{\max}$ 　　　　　　　　　　　　　　　　　　　// size of B in words
5　　R = new BigInteger with all words set to 0.
6　　**for** $i_A = 0$ **to** $s_A - 1$ **by** 1 **do**
7　　　　$c = $ `mulAdd`(**inout** R, i_A, B, 0, $s_{\max} - i_A$, $A[i_A]$)
8　　　　**if** $s_B + i_A < s_{\max}$ **then**
9　　　　　　$R[s_B + i_A] = c$
10　　　　**end**
11　　**end**
12　　**return** R

CHAPTER 7

Results

This chapter is structured in two main parts. The first part (Section 7.1) contains the results from the Java prototype. It includes achieved rendering results as well as measured runtimes of the Java prototype. The second part (Section 7.2) provides runtime measurements and comparisons for the GPU implementation.

7.1 Java Prototype

The performance tests with the Java prototype are executed on a Mac Book Pro (15-inch, 2016) with an 2.9 GHz Intel Core i7. The measured results from these tests are shown in Table 7.4 and Table 7.3. All results are from a single-threaded implementation of the proposed algorithms. The purpose of the Java implementation is to prove the concept and, in its current form, is not performance-optimized. All runtimes shown in Table 7.2 and Table 7.3 are measured with volume size of $100 \times 100 \times 100$ voxels. The rendered image always has a size of 150×150 pixels. Figure 7.1 contains some examples of the images rendered for the performance test.

Table 7.2 shows the runtime performance required for encrypting a volume with scalar voxel values, X-ray rendering, and image decryption with different public key modulus lengths. The table is divided into four groups of rows. The first two groups show the required time for rendering with nearest-neighbor sampling. Group three and four show the resulting performance for trilinear interpolation. The numbers in group one and three of the Table 7.2 are measured without obfuscation during the encryption; therefore, the encrypted volume is not secure. While this type of "encryption" does not have any practical relevance, it is interesting to compare these runtime numbers with those in the group two and four, which are measured from a secure encryption with obfuscation. It can be seen that the obfuscation takes a significant amount of time. Therefore, the random number generation (r) that is required for the obfuscation and the calculation of r^N (see Algorithm 3.2) has a substantial impact on the time required for encrypting

Table 7.1: Required storage size for an encrypted volume with $100 \times 100 \times 100$ voxels and varying modulus length.

	plain (8 bit)	64 bit	128 bit	256 bit	512 bit	1024 bit	2048bit
scalar	1 MB	16 MB	32 MB	64 MB	128 MB	256 MB	512 MB
2 dim	2 MB	32 MB	64 MB	128 MB	256 MB	512 MB	1024 MB
3 dim	3 MB	48 MB	96 MB	192 MB	384 MB	768 MB	1536 MB
4 dim	4 MB	64 MB	128 MB	256 MB	512 MB	1024 MB	2048 MB

(a) nearest-neighbor   (b) trilinear interpolation   (c) transfer function
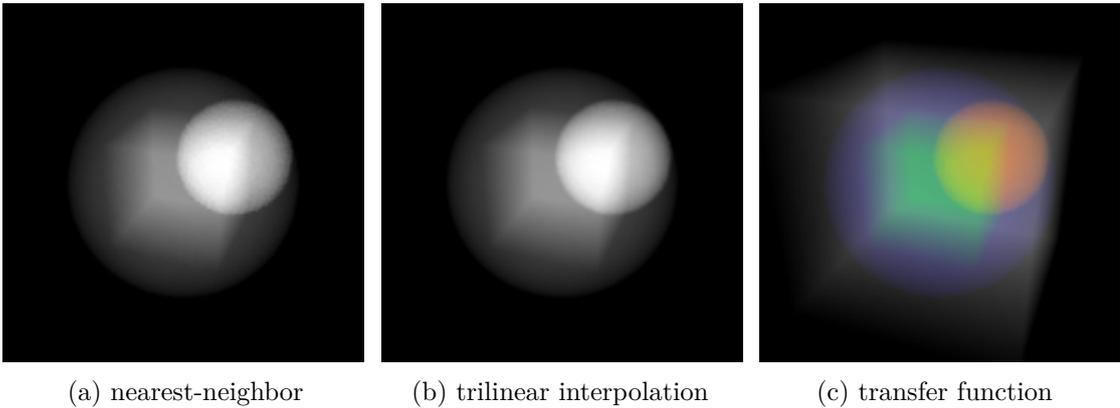
Figure 7.1: Encrypted rendering results; Volume size: $100 \times 100 \times 100$ voxels; Image size: 150×150 pixels (the shown images are scaled by LATEX)

the volume dataset. We use the `java.security.SecureRandom` class from the java standard runtime framework as random number generator for the obfuscation.

Table 7.1 shows the required memory size for this volume with a single scalar value per voxel and also for encodings in multiple dimensions at different modulus length. Table 7.3 shows the runtime required for encrypting a volume with different voxel encodings (two, three, and four dimensions), rendering with our simplified transfer function approach at different counts of *TF-Nodes* (one, two, ... colors) and image decryption. The resulting performance for all these operations is provided for different public key modulus lengths.

Table 7.2: X-ray: Required time (in seconds) for encryption, rendering and decryption with different modulus lengths.

			plain	64 bit	128 bit	256 bit	512 bit	1024 bit	2048 bit
nearest neighbor	without obfuscation	**encrypt**		0.46	0.48	0.54	0.56	0.62	0.57
		render	0.03	0.54	0.66	0.89	1.61	3.49	9.49
		decrypt		0.17	0.25	0.63	2.56	14.92	99.56
	with obfuscation	**encrypt**		5.72	15.36	59.54	327.24	2256.01	16880.00
		render	0.03	1.10	1.77	4.18	11.61	37.10	94.30
		decrypt		0.21	0.43	1.36	4.94	27.24	185.94
trilinear interpolation	without obfuscation	**encrypt**		0.46	0.47	0.52	0.57	0.54	0.65
		render	0.08	10.59	13.55	21.38	47.25	146.07	487.58
		decrypt		0.14	0.26	0.64	2.59	14.65	100.72
	with obfuscation	**encrypt**		5.82	14.67	59.93	330.56	2226.01	16512.47
		render	0.08	16.67	23.86	47.07	121.05	385.71	1182.48
		decrypt		0.20	0.41	1.20	4.89	26.86	186.38

Table 7.3: Simplified transfer function: required time (in seconds) for encryption, rendering and decryption with different modulus length.

			plain	128 bit	256 bit	512 bit	1024 bit	2048 bit
2 dimensional encoding	1 color	encrypt		29.87	115.03	622.77	4405.69	31796.00
		render	0.05	21.85	48.59	142.36	544.35	1999.51
		decrypt		0.80	1.37	4.95	27.7	180.40
	2 colors	render	0.06	23.38	58.20	131.10	659.48	2716.51
		decrypt		0.33	1.18	3.23	27.09	187.51
3 dimensional encoding	1 color	encrypt		40.30	162.22	897.28	6271.48	47030.98
		render	0.05	19.59	40.21	143.69	560.98	2299.65
		decrypt		0.53	0.83	4.47	25.98	127.84
	2 colors	render	0.06	25.87	56.24	154.71	770.64	2850.69
		decrypt		0.43	0.80	3.18	27.67	120.99
	3 colors	render	0.06	41.67	89.73	257.51	919.41	3685.33
		decrypt		0.59	1.21	4.83	25.46	181.86
4 dimensional encoding	1 color	encrypt		54.97	213.21	1199.68	8512.23	62959.05
		render	0.06	19.46	42.93	98.08	429.30	2292.48
		decrypt		0.28	0.77	1.54	8.50	60.03
	2 colors	render	0.07	35.41	61.60	179.45	729.11	3234.38
		decrypt		0.46	0.76	3.10	18.36	121.76
	3 colors	render	0.07	46.41	93.63	268.59	1045.56	4251.69
		decrypt		0.45	1.11	4.52	26.15	190.63-
	4 colors	render	0.08	63.57	123.32	343.20	1320.86	5217.38
		decrypt		0.67	1.51	4.68	26.82	190.49

7.1.1 Images

The images in Figure 7.2 and Figure 7.3 show rendering results of real world datasets and demonstrate what can be done with our simplified transfer function. The dataset from which the images of Figure 7.3 are rendered is an industrial CT scan of a Christmas present that contains a water globe wraped in a box. The images in Figure 7.2 demonstrate the utilization in nuclear medicine. A lung cancer is highlighted by our simplified transfer function approach. During the diagnosis, these datasets are usually investigated either by showing single slices or by X-ray renderings, where the depth cues are provided through rotating the dataset around an axis. This is possible with our homomorphic-encrypted volume rendering with the added privacy, which is useful for diagnosing from such a highly sensitive type of modality and associated pathologies.
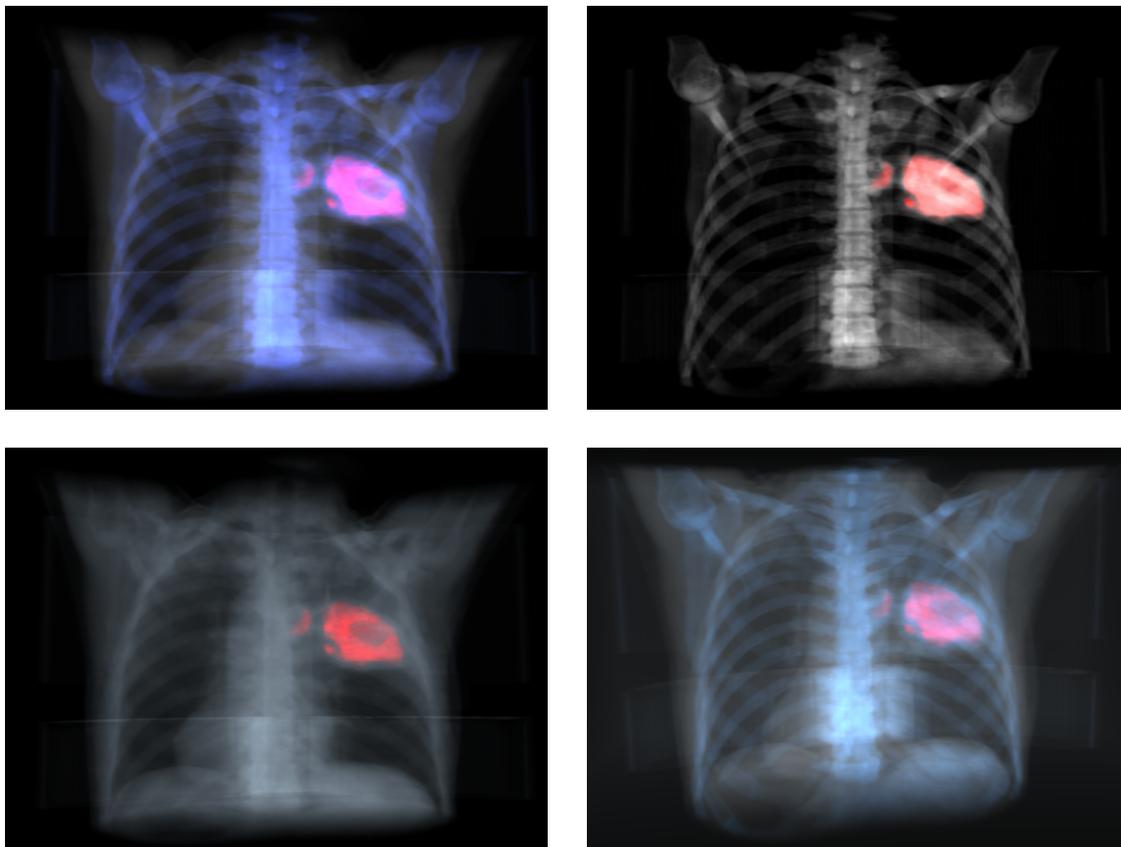
Figure 7.2: The images are rendered by our simplified transfer function approach. Every image was created from the same volume with four-dimensional encoded and encrypted voxel values. The used volume datset is a CT/PET scan of the thorax from a patient with lung cancer (*G0061/Adult-47358/7.0* from the *Lung-PET-CT-Dx* dataset [42, 5]).
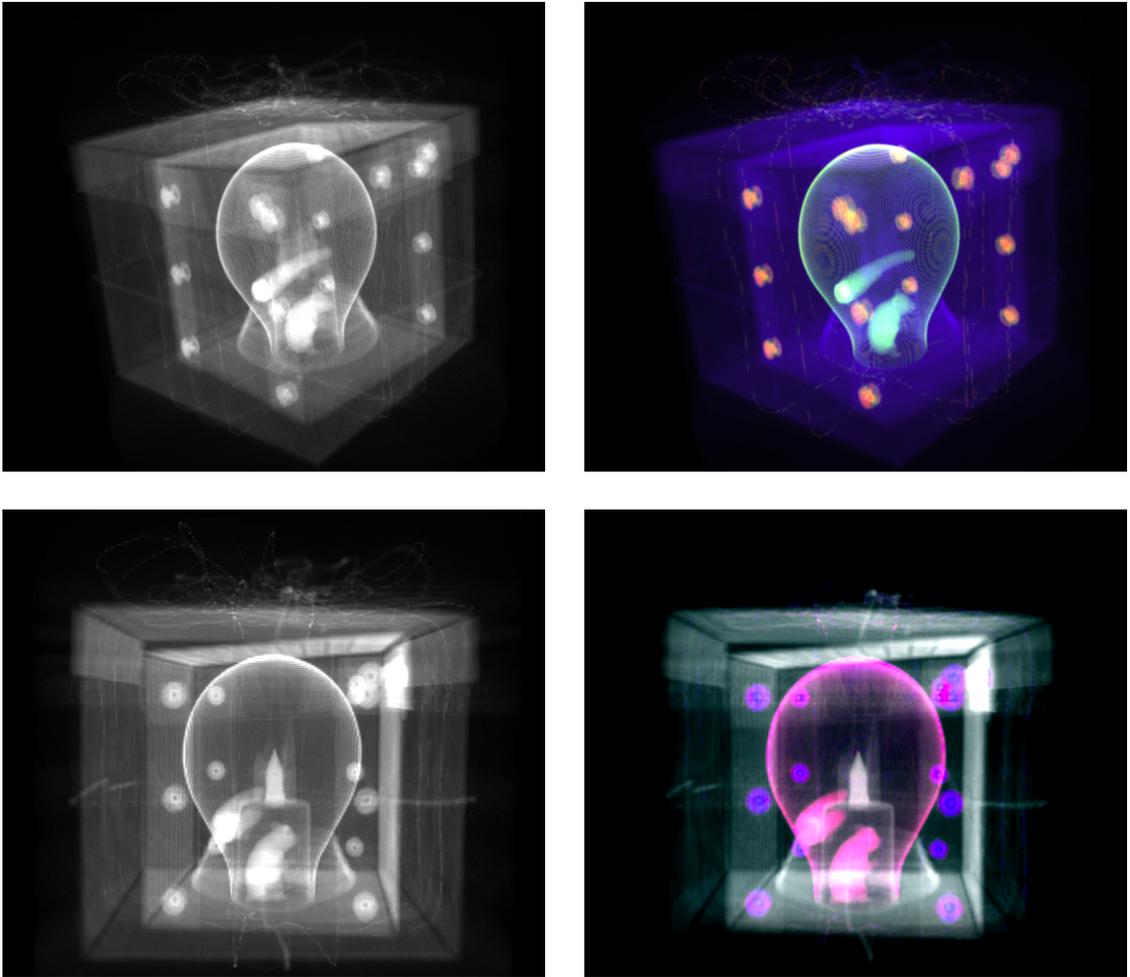
Figure 7.3: The images are rendered from a CT scan of a water globe and other objects inside a box that is wrapped inside a present box [25]. The left images are created by our X-ray approach from a volume with an encrypted scalar value per voxel. The right images are created by our simplified transfer function approach. The volume data voxel values are encoded by four-dimensional vectors. The image on the top and on the bottom differentiate each other not only by a different viewing angle but also by the transfer function used for rendering.

7.2 GPU Performance

The implementation of the rendering algorithms in GLSL can be considered to be written in a more performance friendly way than the rendering code in Java. Since the Java code is written as a playground and proof of concept, it was important that the code is generic, flexible, and easy to implement and not that it is as fast as possible. Therefore, the rendering code implemented in Java is in no way performance optimized. On the other hand most parts of the Vulkan rendering pipeline are either hardware accelerated or performance optimized code from the GPU driver. Also the fragment shader code that performs the actual ray casting should be quite fast compared to the Java code, since it does not have dynamic memory allocation, hardly any data encapsulations or generic wrappers, and much less conditions that handle lots of edge cases compared to the Java code. However, Java big-integer operations did probably receive lots of optimizations during the years. It must be assumed that Java uses many sophisticated techniques that speed up many parts in different methods of Java `java.math.BigInteger` class [57]. Therefore, we can assume that the performance of our big-integer routines cannot compete with that of Java. Hence, we can conclude that the rendering algorithm is faster at the C++/GLSL implementation, but the basic big-integer arithmetic is highly likely more efficient in Java.

7.2.1 Rendering, Volume, and Image Specifications

The Table 7.4 shows measured times for X-ray rendering with nearest-neighbor sampling. The used dataset is the test volume already shown many times in this thesis, for example in Figure 7.1, but for this set of test with a size of $256 \times 256 \times 256$ voxel. The rendered image has a size of 512×512 pixels. Timings for four different processors (two CPUs and two GPUs) and four different public key lengths (64 bit, 128 bit, 256 bit, and 512 bit) are given. Accordingly, the encrypted volumes for the test have a size of 256 MB for the 64 bit key ($256^3 \cdot 64 \cdot 2$ bit), 512 MB for the 128 bit key ($256^3 \cdot 128 \cdot 2$ bit), 1024 MB for the 256 bit key ($256^3 \cdot 256 \cdot 2$ bit), and 2256 MB for the 512 bit key ($256^3 \cdot 512 \cdot 2$ bit). The rendered images in encrypted form have a size of 4 MB for the 64 bit key ($512^2 \cdot 64 \cdot 2$ bit), 8 MB for the 128 bit key ($512^2 \cdot 128 \cdot 2$ bit), 16 MB for the 256 bit key ($512^2 \cdot 256 \cdot 2$ bit), and 32 MB for the 512 bit key ($512^2 \cdot 512 \cdot 2$ bit).

7.2.2 Processor Specifications

The CPU rendering part with the Java prototype contains timings for a quad core Intel i7-6920HQ laptop CPU with 2.9 GHz base clock frequency and a 24 core AMD EPYC 7401P server processor with a base clock frequency of 2.0 GHz. Since both CPUs support twice as many threads as they have real cores, the multithreaded tests are preformed with 8 threads on the quad core Intel i7 and with 48 threads on the 24 core AMD EPYC. This should ensure an optimal utilization of the available resources.

The tested AMD RX 580 has 8 GB of device memory, 2304 stream processors, 1257 MHz base clock frequency and a memory bandwidth of up to 256 GB/s. The theoretical peak

Table 7.4: Required time (in seconds) for multithreaded encrypted X-ray rendering on CPUs and GPUs with different modulus lengths. Volume size: $256 \times 256 \times 256$ voxel, Image size: 512×512 pixels, Sampling: nearest neighbor; The fastest result per public key length of every Processor is printed in **bold**.

				64 bit	**128 bit**	**256 bit**	**512 bit**
CPU - Java	Intel i7-6920HQ 4×2.9GHz	Div.	1 threaded	32.374	54.182	113.297	334.944
			4c/8t	**9.998**	**18.505**	**43.793**	**115.063**
	AMD EPYC 7401P 24×2.0GHz	Div.	1 threaded	35.403	59.858	138.421	418.382
			24c/48t	**2.335**	**3.044**	**6.577**	**19.749**
GPU - Vulkan/GLSL	AMD RX 580 8GB	Div.	mul school	0.115	0.659	6.035	73.201
			mul optimized	0.101	0.505	4.245	66.402
		Montg.	mul school	0.093	0.566	8.899	245.254
			mul optimized	**0.058**	**0.242**	**3.354**	**43.425**
			mul opt. & red. bran.	0.090	0.411	5.036	82.016
	Nvidia RTX 3090 24GB	Div.	mul school	**0.035**	0.810	5.898	59.038
			mul optimized	0.076	0.731	5.076	49.501
		Montg.	mul school	0.148	0.749	5.299	46.973
			mul optimized	0.108	**0.423**	**2.474**	**22.343**
			mul opt. & red. bran.	0.074	0.678	4.433	39.888

performance is 6.2 tera floating-point operations per second (TFLOPs). The second tested GPU is a Nvidia RTX 3090 that has 24GB of device memory, 10496 CUDA Cores, 1395 MHz base clock frequency and a maximal memory bandwidth of 936.2 GB/s. It can theoretically achieve 35.58 TFLOPS. So, from the technical specifications the Nvidia RTX 3090 should be at least four times faster than AMD RX 580, but for our encrypted X-ray rendering it is hardly twice as fast (see Table 7.4 and Figure 7.4).

7.2.3 Multithreaded CPU Tests

The Table 7.4 and Figure 7.4 do not only show runtimes for the GPU, and the single threaded Java code for the CPU, but also runtimes for a multithreaded rendering with Java. The purpose of the multithreaded Java implementation is to show how well the algorithms scale with the count of used CPU cores. The parallelizing strategy that is used for the multithreaded Java implementation works as followed: Before the ray casting starts a thread pool with t threads is created. The variable t represents the number of threads that can be execute in parallel by the CPU. All threads will have an unique ID (i) between 0 and $t-1$, as well as a counter c which is initialized with zero. The
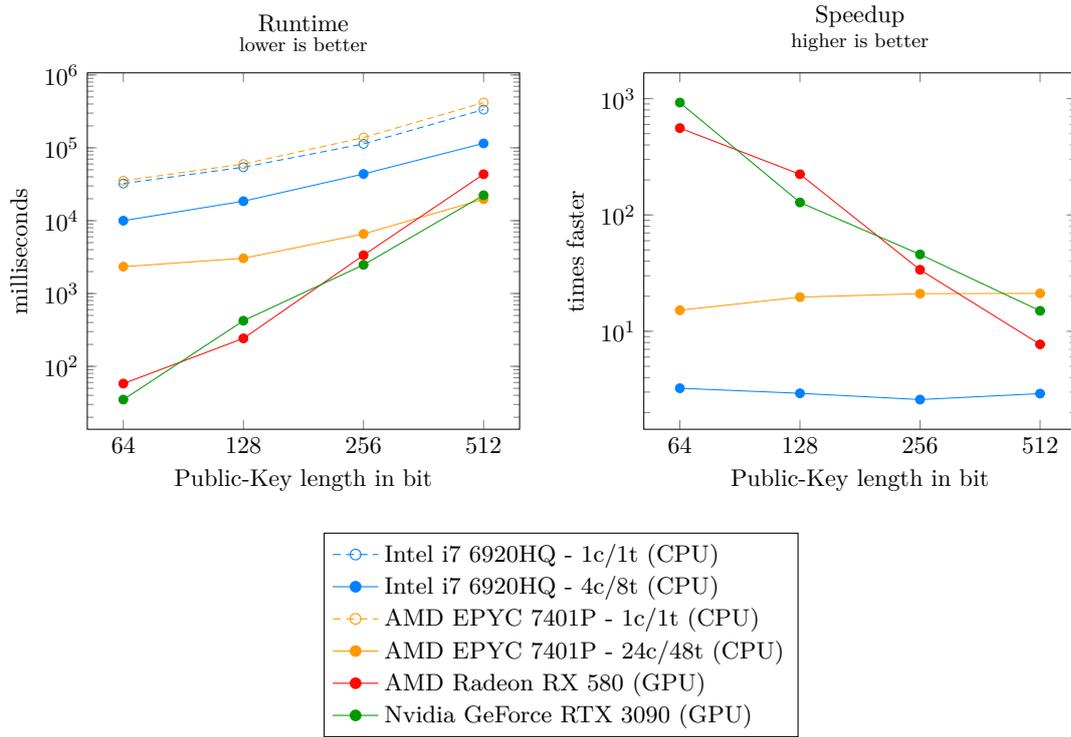
Figure 7.4: Left plot: Required time (in milliseconds) for multithreaded Encrypted X-ray rendering on CPUs and GPUs with different modulus lengths. Right plot: Illustrates the maximal speedup of the parallel implementation for a given processor (sequential runtime divided by parallel runtime). For the GPUs the single threaded runtimes on the Intel i7 were used as sequential runtime reference. The runtimes shown for the GPUs are the results of the algorithm that performs best for a given device and public key length. Volume Size: $256 \times 256 \times 256$ Voxel, Image size: 512×512 Pixels, Sampling: Nearest Neighbor;

ray casting tasks are assigned to the available threads on a per pixel row base. So, one thread calculates one row of the final encrypted image in one round. The rendering of the first row (index 0) of pixels is assigned to the thread with the ID 0 from the thread pool, the second row (index 1) is assigned to the thread with the ID 1 and so on. After a thread has finished with a row it will increase the counter c and start to render the row with the index $i + t \cdot c$. A thread will render rows of pixels until the result of the equation $i + t \cdot c$ is larger than the image height.

The multithreaded speedup on the CPUs depends a little bit on the public key length but not much. On the quad core Intel i7 it is between 2.6 and 3.2. The 24 core AMD CPU achieves speedups between 15.1 and 21.2.

7.2.4 GPU Tests

For every GPU the Table 7.4 contains two groups of rows. One for the runtimes where the classical division is used for the modulo reduction (Div.) and one with results from the Montgomery reduction (Montg.). The lines labeled with "mul school" contain the required runtime for the X-ray rendering when the classical school multiplication (Algorithm 6.7) is used as big-integer multiplication algorithm. For the results stated in lines labeled with "mul optimized" the multiplication with the deferred carry propagation Algorithm 6.8 is used. The runtimes marked as "mul opt. & red. bran." are produced by the multiplication with reduced branching Algorithm 6.10. For every device and each public key length the result of the best performing algorithm is written in bold at the Table 7.4. Hence, it is clearly visible that the fastest algorithm on the GPUs is always the combination of the Montgomery reduction (Section 6.5.4) and the multiplication with the deferred carry propagation (Section 6.5.5), except for the rendering with a 64 bit key on the Nvidia RTX 3090.

The left plot of Figure 7.4 shows the absolute runtimes measured for the CPUs and GPUs at every tested bit length. Two lines are plotted for each CPU. One for the required rendering time with just one thread (dashed) and one line with the result from the tests with multiple threads (solid). For the GPUs only the results of the best performing algorithm at a given key length are drawn. It clearly shows that the GPUs are nearly two magnitudes faster than the 24 core CPU at rendering with a 64 bit short public key. However, the speedup rapidly drops with longer keys. With a 512 bit key the 24 core AMD EPYC even beats the Nvidia GeForce RTX 3090.

The left plot of Figure 7.4 illustrates the achieved speedup for every processor. For the CPUs the speedup is defined as single thread timing divided by the multithreaded result. The base for the speedup calculation of the GPUs is the single threaded result achieved by the Intel i7 with Java which is divided by the result of the best performing algorithm on the GPU (bold values of Table 7.4). The plot with the speedups makes it easy to observe, that the speedup of the CPUs is nearly constant for different public key lengths, while the speedups achieved with the GPUs heavily depend on the length of the public key.

The big question is, why is the speedup from the GPUs so great at calculating with only 256 bit long integers (64 bit key - see Section 6.5.3), but so bad with 2048 bit long integers (512 bit public key). Unfortunately we do not have an answer for that question, we can only guess that it has something to do with the storage size required for holding the ray accumulation buffer big-integer and the current sample value big-integer in the registers and buffers of the GPU. The longer the big-integers get, the harder it gets for the GPU driver and SPIR-V compiler to keep the words required for a multiplication and reduction in the fastest available memories of the GPU. So with longer big-integers more words need to be stored further away from the cores in slower memories, which means more time will be spent on waiting for data.

Discussion

First, we discuss the performance of our implementation and opportunities for improvements. Later, in Section 8.2 starting with general noteworthy considerations, we discuss security-related aspects of our volume rendering approach. Then we discuss some considerations regarding the available precision of floating-point numbers that should be taken into account for an implementation (Section 8.3). In Section 8.4 we follow with an explanation for the invisibility of comparisons. Finally, in the last section of this chapter (Section 8.5), we will show that the used floating-point encoding with an encrypted mantissa and a plaintext exponent does not weaken the privacy of the encrypted volume data.

8.1 Performance

We were unfortunately not able to show interactive frame rates from the presented homomorphic encrypted volume rendering approach, at least not for public key lengths that can be considered to be secure (see Section 8.2). At the Java implementation the rendering code is not efficient and at the GPU implementation the big-integer operations are probably not the fastest possible. The Java implementation, for example, contains multiple unnecessary memory allocations (`new` statements) during the rendering. Also the whole volume storage is not optimal for the voxel access during volume rendering since it is just a naive three-dimensional `java.math.BigInteger` array. A better storage order of voxel values, such as Morton order [52] (recursive Z curve) extended to three dimensions, could lead to a better cache usage, which will improve the performance. Compared to the highly optimized arithmetic algorithms of Java `java.math.BigInteger` class [57] even our improved Montgomery reduction (see Section 6.5.4) is probably inefficient. So even if the rendering pipeline of our GPU implementation should be much faster than the one of the Java prototype, the mathematical functions certainly still have a lot of potential for performance improvement.

However, a major strength of our approach is that it is highly parallelizable and should scale linearly with the processing units. The achieved results form the multithreaded experiments on the CPU with the Java prototype demonstrate this for the rendering part of the approach. Since, the multithreaded CPU rendering prototype was 21 times fasternthe single threaded counterpart on a 24 Core CPU with a 512 bit public key. Not only the rendering but also the encryption and the decryption can be easily implemented as a multithreaded procedure, that scales well with the available procession power since every voxel can be processed independently. That means it should be possible to use as many processing units (e.g., CPU cores or shader hardware on GPU) as voxels in the volume for the encryption. In the rendering and decryption stage, every pixel of the image can be processed independently. Therefore, the number of processing units that can theoretically be used efficiently in parallel is equal to the number of pixels in the final image. While we were able to show impressive speedups of 900 times for the rendering on an Nvidia GeForce RTX 3090 - GPU for 64 bit short keys, we were not able to show efficient usage of the processing power on GPUs for longer public keys. However, there are obvious opportunities to improve our implementation. There are options of algorithmically improvement in the layer of arithmetic operations like the Montgomery multiplication and also possibilities to distribute the rendering workload to more processors. For further reasons on how the performance of our GPU implementation can be improved, we suggest a read of Niall Emmart's Doctoral Dissertation [14] as a starting point. Also a study of Colin Plumb's C library [62] should be advantageous, since it does not only contain a highly optimized modular exponentiation with a sliding window approach (see section "2.2.9.3 Sliding Window Exponentiation" of Niall Emmart's Doctoral Dissertation [14]) and a Montgomery reduction but also lots of valuable comments. Furthermore, our privacy-preserving volume rendering approach should scale much further. It should be possible to use our proposed encrypted voxel compositing scheme as mapper for the MapReduce implementation proposed by Stuart et al. [73], which can make use of a GPU-accelerated distributed memory system for volume rendering.

8.2   Security Considerations

The data privacy of our approach depends entirely on the security of Paillier's cryptosystem. Our approach does not store any voxel value or any information that is computed from a voxel value without an encryption by Paillier's cryptosystem. The Paillier cryptosystem is semantically secure against chosen-plaintext attacks (IND-CPA) [81]. Therefore, we conclude that the data that our approach provides to the storage and rendering server are protected in a semantically secure way. The computational complexity required for breaking a secure key of Paillier's cryptosystem depends on the length of the modulus N. The larger the modulus N is, the harder it is to be factorized, which would be required for data decryption. For the required length of the modulus, the same conditions as for the RSA cryptosystem [66] should hold. From 2018 until 2022, a modulus N with a length of at least 2048 bits is considered to be secure [18, 3].

8.3 Encrypted Number Precision

The Paillier cryptosystem can encrypt and decrypt a value m without an overflow if it is less than the modulus N ($m < N$, see encryption Algorithm 3.2 and decryption Algorithm 3.3). That means for a system that can be considered as secure at least 2^{2023} different numbers can be stored in one encrypted value. The result of 2^{2047} is a number with 617 decimal digits. (https://www.wolframalpha.com) Note that the plaintext values that can be decrypted without an overflow range from 0 to about $2^{\text{bitLength}(N)-1}$, while the encrypted values itself range from 0 to about $2^{\text{bitLength}(N^2)}$. Of course this results in a considerable storage overhead of 512 times for a modulus length of 2048 bit compared to 8 bit voxel value ($2048 \cdot 2/8 = 512$). During an encoding of a standard float or double variable into a mantissa m and an exponent e only the precision that is really required should be used in order to prevent overflows. In other words: during the encoding only as many digits behind the comma as really needed should be stored. While it is possible to decrease an exponent (e.g.: $10^{-10} \to 10^{-11}$) by multiplying the mantissa (see Equation 4.2) of an encrypted floating-point number, it is not possible to increase the exponent (e.g: $10^{-11} \to 10^{-10}$) of an encrypted floating-point number, because this would require a divisions of an encrypted mantissa which is in general not possible. This means that the number with the greatest precision (number with the most digits after the comma) that appears during a calculation also limits the maximal absolute result of the whole calculation.

For example, this is important if the render engine should perform a trilinear filtering (see Section 4.1) , because then we need to calculate the distance between the sample position and the neighboring voxels. If we use double values for the calculation, we could get a number that has a precision of more then 10^{-323} (min. subnormal positive double: $2^{-1022} * 2^{-52} \approx 4.9406564584124654 \times 10^{-324}$ [33, 28]). If such an unnecessarily precise number is used for scaling an encrypted voxel value, the sum of the viewing ray will contain at least 323 digits after the comma until the value is decrypted, because it is not possible to increase the exponent of an encrypted number. If the final sum should be greater than one, the mantissa needs to be at least 323 decimal digits long. Therefore, the plaintext distance (scaling factor) should be rounded to a number with less digits after the comma before multiplying it with the encrypted voxel value in order to prevent wasting 323 decimal digits of the mantissa.

Let us assume a floating-point number system together with a secure- / public-key pair where the mantissa has 600 digits in a decimal representation. In such a system the largest absolute value that can be represented in a sequence of arithmetic operations that contains an intermediate result of 10^{-600} is < 1. However, a mantissa with 600 decimal digit should provide enough precision for many use cases. Take a volume with 10 bits per voxel ($2^{10} = 1024$ possible values) for its unencrypted representation as an example. Even if we calculate the sum of one million (10^6) samples and use a trilinear filtering with a precision of 10^{-3}, we only need a mantissa length of $\log_{10}(10^6 \cdot 2^{10} \cdot 10^3) \approx 12.01 \implies 13$ decimal digits, but for a modulus with a length of 2048 bits we have more than 600 decimal digits available.

8.4   Encrypted Comparison Operators

It is not possible to compare encrypted numbers with each other. During the encryption of a number, the obfuscation is performed (see Section 3.1), which randomly distributes the encrypted values between 0 an $N^2 - 1$. Therefore, the order of the encrypted values $[\![M]\!]$ has nothing to do with the order of the underlying numbers M that were encrypted. Consequently, operators such as lower than ($<$) or greater than ($>$) cannot provide a result that is meaningful for the numbers M, if they are applied to encrypted values $[\![M]\!]$.

We can also argue that comparison operators cannot exist if the Paillier cryptosystem is secure, since the existence of a comparison operator would break the security of the cryptosystem. Consider a less-than comparison for example: if such a comparator could be implemented, every value could be decrypted within $\log_2(N)$ comparisons by a binary search. For a modulus N with a length of 2048 bit, an attacker would need to encrypt and then compare only $\log_2(2^{2048}) = 2048$ numbers with the encrypted value $[\![m]\!]$ in order to find the decrypted number m. This would effectively break the security of the encryption scheme.

8.5   Plaintext Exponent Does Not Leak Private Data

At first glance, it may look like the floating-point representation (encrypted mantissa, plaintext exponent) we used will allow an attacker to obtain more important information than within an encoding where all number components are encrypted. However, if it is implemented correctly, an attacker cannot take any advantage from this number representation. First, we will discuss this for the data in the server memory and, in the last paragraph, we will show how the exponent can be protected during the data transfer from the server to the client.

For the following, we will suppose a secure system with an at least 2048-bit long modulus N and, therefore, a mantissa $[\![m]\!]$ with at least 600 decimal digits usable in the plaintext domain. Voxel values that are stored as 10 bit values are probably precise enough for most volume-rendering use cases. To store numbers between 0 and $2^{10} = 1024$, the exponent e is not required at all, because the voxel information can be stored only in the mantissa m. Therefore, the exponent e can be 1 for all voxels. This means that the exponent does not even have to be transferred to the server, because the server can implicitly assume that the exponents of all numbers is 1. An addition of any of these numbers that have an exponent of 1 does not change the exponent, because for an addition, the exponent needs to be taken into account only if the summands have different exponents (see Algorithm 4.1). Therefore, only a multiplication (e.g., an interpolation between voxel values) can change the exponent to anything other than 1. However, the Paillier cryptosystem only supports the multiplication of an encrypted number with an unencrypted number. Consequently, the number d that changes an exponent has to be unencrypted. Furthermore, this number d can only depend on unencrypted data, because Paillier does not support comparison operators (see Section 8.4), which are required

for flow control statements like `if` or `for-loops`, and arithmetic operations with an encrypted number will result in useless random noise, except those add ($\oplus$) and multiply ($\otimes$) that are defined for the Paillier cryptosystem. Therefore, the number d can only be the result of some computation with other unencrypted variables. This implies that d does not need to be encrypted, because everyone can calculate d itself. In other words, if the variable d can be computed from some variables that need to be considered as publicly available, because they are unencrypted, it is pointless to encrypt d. If d, which is unencrypted and can only depend on unencrypted data, influences an exponent, the exponent exposes only the information that is already publicly available.

The important observation here is that an unencrypted value (e.g., an exponent) can influence an encrypted value (e.g., a mantissa), but an encrypted value (e.g., a mantissa) cannot influence an unencrypted value (e.g., an exponent). This means that no information that is only available as encrypted data can ever be exposed in unencrypted values like the exponent.

In our rendering system, a number d that changes an exponent can either be the result of a computation with a constant or with an unencrypted number that is provided in unencrypted form to the rendering system, such as the camera properties (position of eye point, opening angle, view direction...). Therefore, an attacker could possibly learn the constants used in our program code and data, such as the camera properties that are provided in the unencrypted form, from the exponents of the rendering result (the image). However, we want to develop an approach that is open and semantically secure by design and not *secure through obscurity* (compare: [26, 67, 68, 60]). Therefore, we have to treat the source code of the application as publicly available, which means that a constant cannot be considered to be private. Furthermore, for our approach, the camera properties need to be provided in an unencrypted form to the rendering system. Therefore, we cannot consider it as private anyway.

It should be noted that the camera properties could possibly provide interesting information to an attacker, because it could be possible to learn something about the volume data by tracking the camera properties over time. For instance, if a user rotates the camera around a specific region for a considerable amount of time, an attacker could guess that the region contains some interesting data. During the transfer of the camera properties from the client to the server over the network, the camera properties could be secured by using an encrypted tunnel, such as *IPsec* [31] or *TLS* [64]. However, our basic assumption is that we cannot trust the server that hosts our rendering program. This means that an attacker has access to the entire memory of the server and, therefore, can read the camera properties directly from the memory of the server, regardless of the used network transfer method. While the unencrypted camera properties could indirectly expose some information, we will not discuss this further because it is beyond the scope of this work.

Based on the arguments stated in this section, we can conclude that using plaintext exponents for the rendering process on an untrusted computer system does not provide

more information to a third party than using encryption for all components of a floating-point number.

The only remaining part that needs to be considered is the transfer of the final image from the server back to the client across a network. Operations like trilinear interpolation will change the exponents during the rendering. Therefore, the final image will contain floating-point numbers with exponents unequal to 1 and, because the interpolation weights that change the exponents depend on the camera properties, the exponents of the final image will provide some information about the camera properties. The privacy of the information that is stored in the exponents is only important if it can be assumed that the server is trustworthy, which contradicts the basic assumption of this work. Therefore, this is somewhat beyond the scope of this work, but we nonetheless discuss it here for the sake of completeness. In order to encrypt as much information as possible during the image transfer from the server to the client, ideally all information should be stored in the encrypted mantissa. While it is not possible to divide an encrypted number, it is possible to multiply an encrypted number. Furthermore, the encrypted mantissa can store numbers in the range from 0 to 2^{2047}. Therefore, it is possible to bring all exponents to the value of the smallest exponent of any pixel of the final image. This can be achieved by the calculation shown in Equation 4.2. For the new exponent e_n, the value of the smallest exponent of any pixel must be used. If this exponent-decrease operation is applied to all image values on the server before transferring the image to the client, the exponent should not contain any important information during the transfer, because all exponents then contain the same value. However, if there is concern that even this might contain something useful, it is possible to encrypt this exponent with the public key because the client that has the secure key can decrypt it anyway. Since it is the same value for every number that is sent back to the client, this exponent needs to be sent and decrypted only once.

CHAPTER 9

Conclusions

While the expressiveness of our renderings is far from what is possible with state-of-the-art algorithms for non-encrypted data, we have presented a highly parallelizable direct volume rendering approach that allows not only the outsourcing of the storage of the volume data, but also the outsourcing of the whole rendering pipeline, without compromising the privacy of the data. The approach we propose does not leak any voxel values or any information computed from a voxel value after the volume encryption. Since we encrypt every single bit of voxel data with Paillier's cryptosystem, which is provably semantically secure (see: [58, 81]), it is rather obvious that with our approach, the confidentiality of the volume data (densities, shapes, structures,..) and the colors of the rendered image only depends on the privacy of the secure key. If we trust all devices that have seen the volume data before encryption (e.g.,: MRI-/CT-scanner, the computer that performs the encryption) to safely delete the data after encryption, only the owner of the secure key is able to obtain any useful information of the encrypted volume or rendered images. This is a significant advantage compared to all previous works to date. However, this security naturally comes with associated costs. The storage overhead costs for computation are between four and five orders of magnitude compared to plaintext data.

While we hope that further improvements of our approach would lead to rendering results with better expressiveness, it will be a non-trivial task because the security aspect needs to be considered for even the slightest change. Many of the ideas we considered in the algorithmic design eventually led to a leak of sensitive information, which is, in our opinion, intolerable, no matter how small it may be.

Future work definitively needs to improve the rendering performance. This leads immediately to the open question: Why does the GPU implementation scale so badly for longer public keys compared to the CPU version (see Figure 7.4). We hope that further research leads to an answer for that question and that the current bottleneck of our implementation can be fixed, which should already lead to significant performance improvements. A faster rendering should also be achievable from a more efficient implementation of the

Montgomery multiplication, like the one from Colin Plumb [62]. This should not only help with the encrypted rendering but also with the encryption and decryption process, since it can be used for the modular exponentiation.

The application for encrypted rendering on the GPU, which was implemented in the context of this thesis, currently only supports the basic X-ray rendering and none of the more advanced rendering techniques like the simplified transfer function (Section 5.3) that we introduced. In the scope of this thesis only the implementation within the Java prototype was possible. Therefore, a fast GPU implementation of these techniques is still an interesting problem.

Another possible improvement within the scope of Paillier HE will be the visual quality of compositing. This can be done with gradient-magnitude opacity modulation, where the gradient magnitude will be pre-calculated and encrypted along with the data values. Such representation can already lead to substantial visual quality improvement, although it will still not reach the outcome of compositing using Porter/Duffs's over operator [63]. For the Paillier HE scheme, we do not see a way to implement the over operator compositing, as it requires a multiplication of encrypted numbers. To support alpha blending, new research should be oriented on investigating other homomorphic encryption schemes or a combination of those that, unlike Paillier, would support desired secure alpha blending functionality.

Bibliography

[1] Abbas Acar et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Computing Surveys* 51.4 (2018), 79:2–79:35. DOI: `10.1145/3214303`.

[2] Robert Baillie and Samuel S. Wagstaff. "Lucas Pseudoprimes". In: *Mathematics of Computation* 35.152 (1980), pp. 1391–1417. DOI: `10.1090/S0025-5718-1980-0583518-6`.

[3] Elaine B. Barker et al. *Recommendation for Pair-Wise Key Establishment Using Integer Factorization Cryptography*. Tech. rep. NIST Special Publication 800-56B Rev. 2. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, 2019. DOI: `10.6028/NIST.SP.800-56Br2`.

[4] Jia-Kai Chou and Chuan-Kai Yang. "Obfuscated Volume Rendering". In: *The Visual Computer* 32.12 (2016), pp. 1593–1604. DOI: `10.1007/s00371-015-1143-6`.

[5] Kenneth Clark et al. "The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository". In: *Journal of Digital Imaging* 26.6 (2013), pp. 1045–1057. DOI: `10.1007/s10278-013-9622-7`.

[6] Confidential Computing at CSIRO's Data61. *javallier*. Online. URL: `https://github.com/n1analytics/javallier` (visited on Apr. 17, 2020).

[7] Confidential Computing at CSIRO's Data61. *python-paillier*. Online. URL: `https://github.com/n1analytics/python-paillier` (visited on Apr. 17, 2020).

[8] Timothy J. Cullip and Ulrich Neumann. *Accelerating volume reconstruction with 3D texture hardware*. Tech. rep. University of North Carolina at Chapel Hill, May 1994. URL: `http://www.cs.unc.edu/techreports/93-027.pdf`.

[9] Lee Wen Dick. "Large Integer Arithmetic in GPU for Cryptography". Bachelor's thesis. Universiti Tunku Abdul Rahman, 2017.

[10] Yao Dong, Ana Milanova, and Julian Dolby. "JCrypt: Towards Computation over Encrypted Data". In: *Proceedings of PPPJ*. 2016. DOI: `10.1145/2972206.2972209`.

[11] Yao Dong, Ana Milanova, and Julian Dolby. "SecureMR: Secure Mapreduce Computation Using Homomorphic Encryption and Program Partitioning". In: *Proceedings of HoTSoS*. 2018, pp. 841–848. DOI: `10.1145/3190619.3190638`.

[12]   Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. "Volume Rendering". In: *Proceedings of SIGGRAPH*. 1988, pp. 65–74. DOI: 10.1145/54852.378484.

[13]   Morris J. Dworkin et al. *Advanced Encryption Standard (AES)*. Tech. rep. Federal Information Processing Standards Publication 197. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, 2001. DOI: 10.6028/NIST.FIPS.197.

[14]   Niall Emmart. "A Study of High Performance Multiple Precision Arithmetic on Graphics Processing Units". Doctoral Dissertation. University of Massachusetts Amherst, 2018. DOI: 10.7275/11399986.0.

[15]   Nelly Fazio et al. "Homomorphic Secret Sharing from Paillier Encryption". In: *Proceedings of Provable Security*. 2017, pp. 381–399. DOI: 10.1007/978-3-319-68637-0_23.

[16]   Free Software Foundation, Inc. *GMP - The GNU Multiple Precision Arithmetic Library*. Online. URL: https://gmplib.org (visited on Mar. 29, 2021).

[17]   Craig Gentry. "A fully homomorphic encryption scheme". PhD thesis. Stanford University, 2009.

[18]   Damien Giry and Jean-Jacques Quisquater. *BSI Cryptographic Key Length Report (2018)*. Online. URL: https://www.keylength.com/en/8/ (visited on Mar. 30, 2020).

[19]   Google. *encrypted bigquery client*. Online. URL: https://github.com/google/encrypted-bigquery-client (visited on Apr. 17, 2020).

[20]   Eduard Gröller and Helwig Hauser. "Lecture slides - Volume Visualization". Vienna University of Technology - Faculty of Informatics - Institute of Visual Computing and Human-Centered Technology - Research Division of Computer Graphics. 2008.

[21]   William Harmon. *AMD Radeon RX 6800 16GB GPU Review*. Online. URL: https://www.servethehome.com/amd-radeon-rx-6800-16gb-gpu-review/ (visited on Mar. 29, 2021).

[22]   Owen Harrison and John Waldron. "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware". In: *International Conference on Cryptology in Africa*. 2009, pp. 350–367. DOI: 10.1007/978-3-642-02384-2_22.

[23]   Laszlo Hars. "Modular Inverse Algorithms without Multiplications for Cryptographic Applications". In: *EURASIP Journal on Embedded Systems* 2006.1 (2006), pp. 2–13. DOI: 10.1155/ES/2006/32192.

[24]   Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. third. Alan R. Apt, 2004. Chap. 10, pp. 563–575.

[25]   Christoph Heinzl. *Christmas Present [Dataset]*. 2006. URL: https://www.cg.tuwien.ac.at/research/publications/2006/dataset-present/ (visited on Apr. 1, 2020).

[26] Jaap-Henk Hoepman and Bart Jacobs. "Increased Security through Open Source". In: *Communications of the ACM* 50.1 (2007), pp. 79–83. DOI: `10.1145/1188913.1188921`.

[27] Yin Hu. "Improving the Efficiency of Homomorphic Encryption Schemes". PhD thesis. Worcester Polytechnic Institute, 2013.

[28] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019* (2019), pp. 17–18. DOI: `10.1109/IEEESTD.2019.8766229`.

[29] Robert J. Jenkins. "ISAAC". In: 3rd*Fast Software Encryption Workshop*. 1996, pp. 41–49. DOI: `10.1007/3-540-60865-6_41`.

[30] James T. Kajiya. "The Rendering Equation". In: *Proceedings SIGGRAPH*. 1986, pp. 143–150. DOI: `10.1145/15922.15902`.

[31] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. 2005. URL: `https://www.rfc-editor.org/rfc/rfc4301.txt`.

[32] Auguste Kerckhoffs. "La cryptographie militaire". In: *Journal des sciences militaires* IX (1883), pp. 5–38. URL: `http://www.petitcolas.net/fabien/kerckhoffs/`.

[33] *Wikipedia - Double-precision floating-point format*. Online. URL: `https://en.wikipedia.org/wiki/Double-precision_floating-point_format` (visited on Dec. 26, 2018).

[34] Kitware, Inc. and Contributors. *CMake - Cross Platform Makefile Generator*. Online. URL: `https://cmake.org` (visited on Apr. 5, 2021).

[35] Donald Ervin Knuth. "The art of computer programming". In: 2nd ed. Vol. 2. Addison-Wesley, Reading, MA, US, 1981. Chap. 4.3.1. The Classical Algorithms, p. 250.

[36] Donald Ervin Knuth. "The art of computer programming". In: 2nd ed. Vol. 2. Addison-Wesley, Reading, MA, US, 1981. Chap. 4.5.2 The Greatest Common Divisor, p. 325.

[37] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. DOI: `10.1090/S0025-5718-1987-0866109-5`.

[38] Israel Koren. *Computer Arithmetic Algorithms*. 2nd ed. 2002.

[39] Jens Krüger and Rüdiger Westermann. "Acceleration Techniques for GPU-based Volume Rendering". In: *Proceedings of IEEE VIS*. 2003, pp. 287–292. DOI: `10.1109/VISUAL.2003.1250384`.

[40] Bernhard Langer. "Arbitrary-Precision Arithmetics on the GPU". In: *Central European Seminar on Computer Graphics for Students*. 2015.

[41] Marc Levoy. "Display of surfaces from volume data". In: *IEEE Computer Graphics and Applications* 8.3 (1988), pp. 29–37. DOI: `10.1109/38.511`.

[42]   P. Li et al. *A Large-Scale CT and PET/CT Dataset for Lung Cancer Diagnosis [Dataset]*. The Cancer Imaging Archive, 2020. DOI: `10.7937/TCIA.2020.NNC2-0461`.

[43]   Mian Lu, Bingsheng He, and Qiong Luo. "Supporting Extended Precision on Graphics Processors". In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware.* 2010, pp. 19–26. DOI: `10.1145/1869389.1869392`.

[44]   LunarG, Inc. and Contributors. *Vulkan software development kit (SDK)*. Online. URL: `https://vulkan.lunarg.com/sdk/home` (visited on Apr. 5, 2021).

[45]   Martin Hořeňovský and Contributors. *Catch2 - unit testing framework for C++*. Online. URL: `https://github.com/catchorg/Catch2` (visited on Apr. 5, 2021).

[46]   Paulo Martins, Leonel Sousa, and Artur Mariano. "A Survey on Fully Homomorphic Encryption: An Engineering Perspective". In: *ACM Computing Surveys* 50.6 (2017), 83:1–83:33. DOI: `10.1145/3124441`.

[47]   Sebastian Mazza, Ivan Viola, and Daniel Patel. "Homomorphic-Encrypted Volume Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (2020), pp. 635–644. DOI: `10.1109/TVCG.2020.3030436`.

[48]   Ralph C. Merkle. "Secure Communications over Insecure Channels". In: *Communications of the ACM* 21.4 (1978), pp. 294–299. DOI: `10.1145/359460.359473`.

[49]   Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Proceedings of CRYPTO.* 1986, pp. 417–426. DOI: `10.1007/3-540-39799-X_31`.

[50]   Manoranjan Mohanty, Muhammad Rizwan Asghar, and Giovanni Russello. "3DCrypt: Privacy-Preserving Pre-Classification Volume Ray-Casting of 3D Images in the Cloud". In: *Proceedings of E-Business and Telecommunications.* 2016, pp. 283–291. DOI: `10.5220/0005966302830291`.

[51]   Peter L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. DOI: `10.1090/S0025-5718-1985-0777282-X`.

[52]   G.M. Morton. *A computer Oriented Geodetic Data Base and a New Technique in File Sequencing.* Tech. rep. 150 Laurier Avenue, West, Ottawa 4, Ontario, Canada: IBM Co. Ltd., 1966.

[53]   Andrew Moss, Daniel Page, and Nigel P. Smart. "Toward Acceleration of RSA Using 3D Graphics Hardware". In: *Cryptography and Coding.* Berlin, Heidelberg, 2007, pp. 364–383. ISBN: 978-3-540-77272-9. DOI: `10.1007/978-3-540-77272-9_22`.

[54]   Muhammad Movania et al. "OpenGL – Build high performance graphics". In: 2017. Chap. 7 GPU-based Volume Rendering Techniques, pp. 219–259.

[55] Takatoshi Nakayama and Daisuke Takahashi. "Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing". In: *Proceedings of Parallel and Distributed Computing and Systems*. 2011, pp. 343–349. DOI: `10.2316/P.2011.757-041`.

[56] Project Nayuki. *Montgomery reduction algorithm in Java*. Online. URL: `https://www.nayuki.io/page/montgomery-reduction-algorithm` (visited on Mar. 17, 2021).

[57] OpenJDK. *java.math.BigInteger class*. Online. URL: `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/math/BigInteger.java` (visited on Mar. 17, 2021).

[58] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Procceedings of EUROCRYPT*. 1999, pp. 223–238. DOI: `10.1007/3-540-48910-X_16`.

[59] Andrew Paverd, Andrew Martin, and Ian Brown. "Modelling and Automatically Analysing Privacy Properties for Honest-but-Curious Adversaries". Online. 2014. URL: `https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf` (visited on July 30, 2019).

[60] Chad Perrin. *Security 101, Remedial Edition: Obscurity is not security*. Online. URL: `https://www.techrepublic.com/blog/it-security/security-101-remedial-edition-obscurity-is-not-security/` (visited on Apr. 17, 2020).

[61] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Communications of the ACM* 18.6 (1975), pp. 311–317. DOI: `10.1145/360825.360839`.

[62] Colin Plumb. *bnlib - The BigNum multi-precision integer math library*. Online. URL: `https://www.schneier.com/wp-content/uploads/2015/03/BNLIB11-2.zip` (visited on Mar. 17, 2021).

[63] Thomas Porter and Tom Duff. "Compositing Digital Images". In: *Proceedings of SIGGRAPH*. 1984, pp. 253–259. DOI: `10.1145/964965.808606`.

[64] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018. URL: `https://www.rfc-editor.org/rfc/rfc8446.txt`.

[65] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation, Academia Press* 4.11 (1978), pp. 169–179.

[66] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126. ISSN: 0001-0782. DOI: `10.1145/359340.359342`.

[67] Karen Scarfone, Wayne Jansen, and Miles Tracy. *Guide to General Server Security*. Tech. rep. NIST Special Publication 800-123. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, 2018. DOI: `10.6028/NIST.SP.800-123`.

[68]   Bruce Schneier. *The Insecurity of Secret IT Systems*. Online. URL: https://www.
         schneier.com/blog/archives/2014/02/the_insecurity_2.html
         (visited on Apr. 17, 2020).

[69]   Inc. Silicon Graphics. *gluLookAt method documentation*. Online. URL: https:
         //www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/
         gluLookAt.xml (visited on Dec. 23, 2018).

[70]   Tomasz Sowa. *TTMath - Bignum C++ library*. Online. URL: https://www.
         ttmath.org (visited on Mar. 17, 2021).

[71]   National Institute of Standards and Technology. *Digital Signature Standard (DSS)*.
         Tech. rep. Federal Information Processing Standards Publication 197. U.S. Depart-
         ment of Commerce, National Institute of Standards and Technology, Gaithersburg,
         MD, 2013. DOI: 10.6028/NIST.FIPS.186-4.

[72]   Julian James Stephen et al. "Practical Confidentiality Preserving Big Data Analysis".
         In: *Proceedings of HotCloud*. 2014.

[73]   Jeff A. Stuart et al. "Multi-GPU Volume Rendering Using MapReduce". In: *Pro-
         ceedings of ACM HPDC*. 2010, pp. 841–848. DOI: 10.1145/1851476.1851597.

[74]   The Khronos Group, Inc. *VkPhysicalDeviceLimits(3) Manual Page*. Online. URL:
         https://www.khronos.org/registry/vulkan/specs/1.2-extensions/
         man/html/VkPhysicalDeviceLimits.html (visited on Apr. 14, 2021).

[75]   The Khronos Group, Inc. *Vulkan - graphics and compute API*. Online. URL: https:
         //www.khronos.org/vulkan/ (visited on Apr. 5, 2021).

[76]   Ivan Viola and Meister Eduard Gröller. "Smart Visibility in Visualization". In:
         Proceedings of Computational Aesthetics in Graphics, Visualization and Imaging.
         2005, pp. 209–216. DOI: 10.2312/COMPAESTH/COMPAESTH05/209-216.

[77]   Markus Stefan Wamser, Stefan Rass, and Peter Schartner. "Oblivious Lookup-
         Tables". In: *Tatra Mountains Mathematical Publications* 67.1 (2016), pp. 191–203.
         DOI: 10.1515/tmmp-2016-0039.

[78]   Blake Warner. *A C++ template based statically allocated arbitary length big integer
         library designed for OpenCL*. Online. URL: https://github.com/blawar/
         biginteger (visited on June 9, 2020).

[79]   *Wikipedia - Trilinear interpolation*. Online. URL: https://en.wikipedia.org/
         wiki/Trilinear_interpolation (visited on Mar. 1, 2019).

[80]   Sascha Willems. *Vulkan Hardware Database*. Online. URL: https://vulkan.
         gpuinfo.org (visited on Apr. 14, 2021).

[81]   Xun Yi, Russell Paulet, and Elisa Bertino. "Homomorphic Encryption and Ap-
         plications". In: Springer, 2014. Chap. 2 Homomorphic Encryption, p. 41. DOI:
         10.1007/978-3-319-12229-8.

[82]   Kaiyong Zhao and Xiaowen Chu. "GPUMP: A Multiple-Precision Integer Library
         for GPUs". In: *IEEE International Conference on Computer and Information
         Technology* (2010), pp. 1164–1168. DOI: 10.1109/CIT.2010.211.

98

[83]   M. Tarek Ibn Ziad et al. "CryptoImg: Privacy preserving processing over encrypted images". In: *Procceedings of IEEE Communications and Network Security*. 2016, pp. 570–575. DOI: 10.1109/CNS.2016.7860550.

Supplementary Algorithms

A.1 Perspective Viewing Ray

How to calculate a viewing ray for a perspective projection is stated in Algorithm A.1. The algorithm calculates the origin ($\vec{o}$) and direction ($\vec{d}$) of a viewing ray for a specific pixel (x, y) of the final image with width (w) and height (h). The camera position, view direction, and up vector are encoded in the matrix M. The matrix M can be calculated, for example, by the function `gluLookAt` of OpenGL Utility Library [69]. The field of view (f) is the opening angle of the camera.

Algorithm A.1: Viewing Ray

Parameters: Image width (w) and height (h), image pixel position x and y for which the ray should be created, the view matrix $M \in \mathbb{R}^{4 \times 4}$ that contains the position and orientation of the camera (e.g. the result of GLMs lookAt() function) and the field of view f.

Result: The ray origin $\vec{o}$ and direction $\vec{d}$.

---

1 **procedure** viewingRay(**in** w, **in** h, **in** x, **in** y, **in** M, **in** f)
2 $\quad$ $r = w/h$ $\hfill$ /* aspect ratio */
3 $\quad$ $s = \tan(f \cdot 0.5)$ $\hfill$ /* scale */
4 $\quad$ $d_x = (2 \cdot (x + 0.5)/w - 1) \cdot r \cdot s$
5 $\quad$ $d_y = (1 - 2 \cdot (y + 0.5)/h) * s$
6 $\quad$ $d_z = -1$
7 $\quad$ $\vec{d} = $ mulDirection($[d_x, d_y, d_z]$, M^{-1})
8 $\quad$ $\vec{o} = $ mulPosition($[0, 0, 0]$, M^{-1})
9
10 **procedure** mulDirection(**in** $\vec{v}$, **in** M)
11 $\quad$ $r_0 = M_{0,0} \cdot v_0 + M_{1,0} \cdot v_1 + M_{2,0} \cdot v_2$
12 $\quad$ $r_1 = M_{0,1} \cdot v_0 + M_{1,1} \cdot v_1 + M_{2,1} \cdot v_2$
13 $\quad$ $r_2 = M_{0,2} \cdot v_0 + M_{1,2} \cdot v_1 + M_{2,2} \cdot v_2$
14
15 **procedure** mulPosition(**in** $\vec{d}$, **in** M)
16 $\quad$ $r_0 = M_{0,0} \cdot v_0 + M_{1,0} \cdot v_1 + M_{2,0} \cdot v_2 + M_{3,0}$
17 $\quad$ $r_1 = M_{0,1} \cdot v_0 + M_{1,1} \cdot v_1 + M_{2,1} \cdot v_2 + M_{3,1}$
18 $\quad$ $r_2 = M_{0,2} \cdot v_0 + M_{1,2} \cdot v_1 + M_{2,2} \cdot v_2 + M_{3,2}$

A.2 Big-Integer Utility Procedures

The following algorithms state the details about utility procedures references from big-integer algorithms in Chapter 6.

Algorithm A.2: Procedure that adds two single word integers and stores the sum in a single word integer and a carry flag. (Based on TTMath [70])

Parameters: The integers a and b that should be summed up and the last carry flag c (that can be 0 or 1).

Result: The sum r as last out parameter and the new carry flag as return value.

1 procedure addTwoWords (**in** a, **in** b, **in** c, **out** r)

2   **if** $c = 0$ **then**

3 $r = a + b$

4   **if** $r < a$ **then**

5 $c = 1$

6   **end**

7   **else**

         // $c = 1$

8 $r = a + b + c$

9   **if** $r > a$ **then**

10 $c = 0$

11   **end**

12   **end**

13   **return** c

Algorithm A.3: Procedure that adds a double word integer to a big-integer magnitude array. (Based on TTMath [70])

Parameters: The lower (least significant) word a' and the higher (most significant) word a'' of the two words long integer that should be added to the big-integer magnitude array T with a length of n. The index p specifies the starting position in T from where the integer a should be added. The equivalent in a decimal system (word size equals 10) can be expressed as: $T = T + a \cdot 10^p$.

Result: The updated big-integer magnitude array T as in/out parameter and the new carry flag as return value.

---

1 procedure addTwoInts (**in** a'', **in** a', **in** p, **inout** T, **in** n)

2   assert $p < (n - 1)$

3 $c =$ addTwoWords $(T[p], a', 0, \textbf{out } T[p])$

4 $c =$ addTwoWords $(T[p + 1], a', c, \textbf{out } T[p + 1])$

5   **for** $i = p + 2$ **to** $n - 1$ **by** 1 **while** $c > 0$ **do**

6 $c =$ addTwoWords $(T[i], 0, c, \textbf{out } T[i])$

7   **end**

8   **return** c

Algorithm A.4: Procedure that adds a single word integer to a big-integer magnitude array. (Based on TTMath [70])

Parameters: The single word integer a that should be added to the big-integer magnitude array T with a length of n. The index p specifies the starting position in T from where the integer a should be added. The equivalent in a decimal system (word size equals 10) can be expressed as: $T = T + a \cdot 10^p$.

Result: The updated big-integer magnitude array T as in/out parameter and the new carry flag as return value.

---

1 procedure addInt (**in** a, **in** p, **inout** T, **in** n)

2   assert $p < n$

3 $c =$ addTwoWords $(T[p], a, 0, \textbf{out } T[p])$

4   **for** $i = p + 1$ **to** $n - 1$ **by** 1 **while** $c > 0$ **do**

5 $c =$ addTwoWords $(T[i], 0, c, \textbf{out } T[i])$

6   **end**

7   **return** c

Algorithm A.5: Procedure that multiplies two single word integers into a double word integer. (Based on TTMath [70])

Parameters: The integers a and b that should be multiplied. Both are a single word long. The subscript x_{low} indicates the access of the lower (least significant) half word and the subscript x_{high} indicates the access of the upper (most significant) half word.

Result: The lower (least significant) word r' and the higher (most significant) word r'' of the two words long result.

---

1 **procedure** `mulTwoWords` (**in** a, **in** b, **out** r'', **out** r')

2 $\quad h' = h'' = l' = l'' = 0$

3 $\quad l' = b_{\text{low}} \cdot a_{\text{low}}$

4 $\quad l'_{\text{high}} = (l'_{\text{high}} + b_{\text{low}} \cdot a_{\text{high}})_{\text{low}}$

5 $\quad h'_{\text{low}} = (l'_{\text{high}} + b_{\text{low}} \cdot a_{\text{high}})_{\text{high}}$

6 $\quad t = b_{\text{high}} \cdot a_{\text{low}}$

7 $\quad l''_{\text{high}} = t_{\text{low}}$

8 $\quad h'' = b_{\text{high}} \cdot a_{\text{high}} + t_{\text{high}}$

9 $\quad c = \text{addTwoWords}\,(l', \ l'', \ 0, \ r')$ // Compute the lower word r' from l' & l''.

10 $\quad \text{addTwoWords}(h', h'', c, r'')$ // Compute the higher word r'' from h' & h''. No carry.

Algorithm A.6: Procedure that multiplies a big-integer with a single word integer. (Based on TTMath [70])

Parameters: The big-integer array A which will be multiplied by the single word integer k.

Result: The big-integer product $R = A \cdot k$ as return value.

1 **procedure** mulInt(**in** A, **in** k)

2   **if** $k = 0$ **then**

3   **return** BigInteger(0);   // short cut, early exit ($A \cdot 0 \implies 0$)

4   **end**

5 $s_{\max}$ = The maximal word count of a BigInteger

6 s_A = wordLength(A)   // size of A in words

7 b_A = findLowestSetWord(A)   // begin of A in words; can be hardcoded to 0

8 R = new BigInteger with all words set to 0.

9   **for** $i_A = b_A$ **to** $s_A - 1$ **by** 1 **while** $i_A < s_{\max} - 1$ **do**

10   mulTwoWords($A[i_A]$, k, **out** r'', **out** r')

11   addTwoInts(r'', r', i_A, **inout** R, $s_{\max}$);

12   **end**

13   **if** $s_A = s_{\max}$ **then**   // multiply with last word if required

14   mulTwoWords($A[s_{\max} - 1]$, k, **out** r'', **out** r')

15   **if** $r'' > 0$ **then**

16   Error: Multiplication not possible without overflow.

17   **end**

18 c = addInt(r', $s_{\max} - 1$, **inout** R, $s_{\max}$)

19   **if** $c > 0$ **then**

20   Error: Multiplication not possible without overflow.

21   **end**

22   **end**

23   **return** R

Matrices for Oblivious Lookup Tables

In the following I will show some of the matrix creation schemes I developed for the Oblivious Lookup Tables (see Section 5.1). In the special context of a transfer function for volume rendering, they can be used as a more efficient alternative than the Vandermonde-Matrix suggested by Wamser et al. [77]. For a lookup table that should only be used as a transfer function it is possible to add a restriction to X that could help to find an approach for expanding an encrypted x_i to an encrypted $\vec{v}_i$. A transfer function always maps an increasing sequence of integer numbers from 0 to $2^r - 1$ where r is the resolution of a voxel in bits, therefore, X can be defined as $X = \{0, 1, 2, ..., 2^r - 1\}$. With this assumption that X contains an increasing sequence $X = \{0, 1, 2, ..., n\}$ it is possible to define $\vec{v}_i$ as:

$$\vec{v}_i = (x_i + 1 \mod n, \quad x_i + 2 \mod n, \quad ..., \quad x_i + n \mod n) \qquad \text{(B.1)}$$

The same result can be calculated even more efficient on binary number representation by using bitwise and (&) operation instead of modulo operations:

$$\vec{v}_i = (x_i + 1 \ \& \ n - 1, \quad x_i + 2 \ \& \ n - 1, \quad ..., \quad x_i + n \ \& \ n - 1) \qquad \text{(B.2)}$$

Unfortunately, this can also not be calculated on encrypted numbers, because the bitwise *and* operation (&) can not be evaluated in the encrypted domain.

For an n that is equal to a power of two result ($n = 2^x$) the Equation B.2 produces the same matrix V as the definition of $\vec{v}_i$ stated in Equation B.1. For $n = 8$ the matrix will look like:

$$
V = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 \\
2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 \\
3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\
4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
5 & 6 & 7 & 0 & 1 & 2 & 3 & 4 \\
6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 \\
7 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{pmatrix}
\tag{B.3}
$$

While it is not possible to calculate modulo for an arbitrary modulus, there is one single residue class for which a modulo operation can be performed in the encrypted domain. This residue class is N which is the modulus of the public key. How to calculate $m \mod N$ in the encrypted domain is shown in Equation B.4.

$$
m \mod N = \mathrm{Dec}(\llbracket m_1 \rrbracket \mod N^2)
\tag{B.4}
$$

In order to create a matrix that has zeros on the diagonal and numbers greater than zero everywhere else, like the matrix shown in Equation B.3, we can scale the values by $s_{\mathbb{R}} = N/n$ before calculating $\mod N$. This leads to the following creation scheme:

$$
\vec{v_i} = ((x_i + 1) \cdot s_{\mathbb{R}} \mod N, \quad (x_i + 2) \cdot s_{\mathbb{R}} \mod N, \quad ..., \quad (x_i + n) \cdot s_{\mathbb{R}} \mod N)
\tag{B.5}
$$

For a 16 bit modulus $N = 45649$ ($= p \cdot q$, with $p = 239$ and $q = 191$) the resulting matrix will look like:

$$
V = \begin{pmatrix}
5706 & 11412 & 17118 & 22825 & 28531 & 34237 & 39943 & 0 \\
11412 & 17118 & 22825 & 28531 & 34237 & 39943 & 0 & 5706 \\
17118 & 22825 & 28531 & 34237 & 39943 & 0 & 5706 & 11412 \\
22825 & 28531 & 34237 & 39943 & 0 & 5706 & 11412 & 17118 \\
28531 & 34237 & 39943 & 0 & 5706 & 11412 & 17118 & 22825 \\
34237 & 39943 & 0 & 5706 & 11412 & 17118 & 22825 & 28531 \\
39943 & 0 & 5706 & 11412 & 17118 & 22825 & 28531 & 34237 \\
0 & 5706 & 11412 & 17118 & 22825 & 28531 & 34237 & 39943
\end{pmatrix}
\tag{B.6}
$$

This matrix is regular in $\mathbb{R}$ and in $\mathbb{Z}_N$. However, it is not possible to multiply an encrypted value m with $s_{\mathbb{R}} = N/n$, because N is only dividable by p and q (the private key values) but not by n. Nonetheless, $s_{\mathbb{R}} = N/n$ can be calculated on the server in plaintext, because it does not contain any private information. Therefore, it is possible

to calculate the rounded value $s_\mathbb{Z} = \lfloor N/n \rceil$. If we replace the $s_\mathbb{R}$ with $s_\mathbb{Z}$ in Equation B.5, we will get the following matrix:

$$V = \begin{pmatrix} 5706 & 11412 & 17118 & 22824 & 28530 & 34236 & 39942 & 45648 \\ 11412 & 17118 & 22824 & 28530 & 34236 & 39942 & 45648 & 5705 \\ 17118 & 22824 & 28530 & 34236 & 39942 & 45648 & 5705 & 11411 \\ 22824 & 28530 & 34236 & 39942 & 45648 & 5705 & 11411 & 17117 \\ 28530 & 34236 & 39942 & 45648 & 5705 & 11411 & 17117 & 22823 \\ 34236 & 39942 & 45648 & 5705 & 11411 & 17117 & 22823 & 28529 \\ 39942 & 45648 & 5705 & 11411 & 17117 & 22823 & 28529 & 34235 \\ 45648 & 5705 & 11411 & 17117 & 22823 & 28529 & 34235 & 39941 \end{pmatrix} \quad \text{(B.7)}$$

While the matrix in Equation B.7 is still regular in $\mathbb{R}$ it is not invertable in $\mathbb{Z}_N$ and, therefore, useless for the Oblivious Lookup Tables.