UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Higher Polytechnic School of Gandia

Volume Rendering for Scientific Visualization

End of Degree Project

Bachelor's Degree in Interactive Technologies

AUTHOR: Brito Clavijo, Abidán

Tutor: Palanca Cámara, Javier

ACADEMIC YEAR: 2023/2024

Abstract

Volume visualization enables extracting meaningful information from volumetric data, using computer graphics and imaging techniques. This text explores a rendering pipeline that allows the representation and exploration of such datasets.

The idea is to create an application that can read in this type of data, load it onto the graphics card, and visualize it interactively. This generally involves the use of transfer functions to map RGBa color tuples to the discrete values of the dataset, as well as the simulation of a virtual environment.

Such software programs are especially useful in the biomedical field. However, they offer applications in numerous other fields, such as geology, archeology, material science (for quality control) and computational science and engineering.

Keywords: scientific visualization, medical imaging, computer graphics, parallel programming, shaders.

Resumen

La visualización de volúmenes permite extraer información significativa de datos volumétricos, utilizando técnicas de gráficos por ordenador y procesamiento de imágenes. En este texto se explorará un pipeline de renderizado que posibilita la representación y exploración de tales conjuntos de datos.

La idea es realizar una aplicación que permita leer este tipo de datos, cargarlos en la tarjeta gráfica y, haciendo uso de técnicas propias de informática gráfica, visualizarlos de forma interactiva. Esto supone la utilización de funciones de transferencia para mapear tuplas de color RGBa a los valores discretos del conjunto, así como la simulación de un entorno virtual.

Este tipo de programas informáticos son especialmente útiles en el campo biomédico, siendo comunes entre radiólogos y otros especialistas. No obstante, ofrecen soluciones extrapolables a numerosos campos, como la geología, la arqueología, la ciencia de materiales (para el control de calidad) y la ciencia e ingeniería computacional.

Palabras clave: visualización científica, imagen médica, informática gráfica, programación paralela, shaders.

Resum

La visualització de volums permet extraure informació significativa de dades volumètriques, utilitzant tècniques de gràfics per ordinador i processament d'imatge. En aquest text s'explorarà un pipeline de renderització que possibilita la representació i exploració d'aquests conjunts de dades.

L'objectiu és realitzar una aplicació que permeta llegir aquest tipus de dades, carregar-les en la targeta gràfica i, fent ús de tècniques pròpies de la informàtica gràfica, visualitzar-les de manera interactiva. Això suposa la utilització de funcions de transferència per mapar tuples de color RGBa als valors discrets del conjunt, així com la simulació d'un entorn virtual.

Aquest tipus de programes informàtics són especialment útils en el camp biomèdic, sent comuns entre radiòlegs i altres especialistes. No obstant això, ofereixen solucions extrapolables a nombrosos camps, com la geologia, l'arqueologia, la ciència de materials (per al control de qualitat) i la ciència i enginyeria computacional.

Paraules clau: visualització científica, imatge mèdica, informàtica gràfica, programació parallela, shaders.

Acknowledgements

First and foremost I would like to thank my advisor, Prof. Dr. Javier Palanca Cámara, for his constant encouragement and feedback throughout the development of the project. Most of all, I am grateful for his trust and for giving me the opportunity to go down the rabbit hole, studying a topic I find deeply fascinating.

I would like to thank my parents for their unconditional support, their love and their constant belief in my abilities. It has proven to be an invaluable source of strength.

I am also grateful to my friends and colleagues for their understanding and patience during times when I was unable to spend time with them.

Lastly, I would like to dedicate this work to my grandmother, who, at 93 years of age, made it a point to come by every other morning for a quick coffee, to check on me.

Contents

List of Figures

Chapter 1

Introduction

In the last couple decades, significant advancements in computing technology have revolutionized the field of volume visualization, once confined to specialized medical institutions and universities, due to its intensive computational requirements. With the proliferation and availability of multi-core microprocessor architectures and, more importantly, the development of dedicated Graphics Processing Units, several methods have been devised to take advantage of massive parallel code execution. As a result, interactive and high-quality volume rendering has become commonplace across various scientific disciplines. This transformation has been fueled, in part, by the widespread adoption of medical imaging technologies like Computed Tomography (CT), which now facilitate easier and more cost-effective acquisition of volumetric data.

Consequently, there has been a surge in demand for sophisticated visualization software capable of leveraging hardware-accelerated rendering and enhanced user interaction. This has led to both full-scale commercial and open-source solutions, such as Inviwo, MeVisLab, OsiriX DICOM Viewer, The Visualization Toolkit (VTK) or InVesalius.

These solutions are especially useful in the medical field, being common among radiologists and other specialists. However, they also offer applications to other areas, such as material and environmental sciences and fluid dynamics. As an example, oil explorations tend to rely on volume visualizations of geoseismic data to pinpoint the right spot for perforation.

1.1 Motivation

The idea of turning complex information into visible form has always been of human interest and employed for millennia, dating back to the Upper Paleolithic era, as illustrated in Fig. 1.1.

Leonardo Da Vinci's sketches (see Fig. 1.1) showcase his exploration of the human body and the world, and while the basis strategy may differ greatly compared to the ones described in this text, the underlying goal remains the same: to convey/portray information clearly and effectively through graphical means.

(a) At least 40,800 years old. (b) Circa 1489.

Figure 1.1: (a): Stag depiction in Cave of the Castle, in Cantabria, Spain. (b): ”View of a skull”, by Leonardo Da Vinci.

However, this process does not entirely align with the modern concept of visualization. As it stands, it is no longer only about documenting existing knowledge, but also a way to gain new one. Visualizations are key for the comprehension and interpretation of scientific datasets. They are used as a means to gain insight and better understanding over the data in question.

Even though data typically has inherent structures, there are often specific ways in which it is most naturally explored. To achieve this, several approaches have been proposed: classification according to a subset of features, different rendering primitives (points, triangles, lines and voxels), projection methods (backwards and forwards), rendering styles (photorealistic and non-photorealistic) and even new user interaction metaphors for improved Focus+Context [1].

1.2 Proposal and objectives

This work aims to address these challenges by developing a desktop software application capable of interactive, high-quality volume rendering at real-time frame rates, with a special emphasis on medical imaging data. Our research topics will include the exploration of GPU-based volume rendering strategies, appropriate data classification and several optimization techniques.

Breaking down the main objective into smaller, more manageable tasks can help shape a clear working direction:

- Study and analyze state of the art in real-time volume rendering.
- Understand common medical imaging formats.
- Learn modern software development practices and standard industry tools.
- Get acquainted with the programmable graphics pipeline.
- Learn a graphics API.

- Familiarization with graphics debuggers.
- Validation and usability testing.

1.3 Document outline

- Chapter 2: **State of the art**
  Volume rendering is a broad and well-researched field. This chapter gives a brief overview of the notions behind it, and references some seminal academic publications.
- Chapter 3: **Problem statement and requirements**
  This chapter outlines the specific problem/s addressed by this work, listing out a number of relevant use cases.
- Chapter 4: **Design**
  Here, we explore the methodology, design choices and rationale behind the application; devising both UI/UX and software development strategies.
- Chapter 5: **Implementation**
  This chapter delves into the ins and outs of the development process: technologies used, in-depth algorithm descriptions, relevant code snippets and challenges faced.
- Chapter 6: **Results and evaluation**
  This chapter presents and analyzes the outcome, comparing images side by side and covering usability concerns.
- Chapter 6: **Conclusions**
  Finally, we summarize our findings, discuss their implications, note any limitations and suggest future lines of improvement and research.

Chapter 2

State of the art

Initially, volume datasets were explored by viewing consecutive data slices. Soon after, attempts were made to extract three-dimensional structures [2]. While early research focused on iso-surface extraction [3], the introduction of Direct Volume Rendering by Drebin et al. [4] and Levoy [5] first demonstrated the advantages of rendering from the volumetric representation itself, i.e., their inner structure.

Since then, a vast amount of research has been devoted to this area. In 1993, Cullip and Neumann pioneered the first hardware-accelerated object-order direct volume rendering algorithm [6], which involves rendering slices of the volume using proxy geometry. Shear-warp factorization, by Lacroute and Levoy [7], presented a fast CPU method that also traversed the volume in a slice-by-slice fashion, but transformed it into a sheared space prior to applying axonometric projection onto a base plane and warping it for image generation. Krüger et al. [8] built upon these advancements, further improving the efficiency and quality of GPU-based volume rendering, by implementing raycasting in the programmable graphics pipeline, while also integrating acceleration techniques.

Volume raycasting, sometimes referred to as volume raymarching, has remained a staple in real-time volume rendering since its inception, with a lot of the academic effort going towards optimized spatial data structures to store and index voxels, automatic transfer functions and hybrid rendering strategies. In recent years, new paradigms have emerged, namely radiance fields and 3D Gaussian Splatting [9], prompted by the Novel View Synthesis and 3D Reconstruction communities. NeRF [10] presented a differentiable volume rendering formula to train a coordinate-based multilayer perceptron in order to directly predict color and opacity values. Plenoxels [11] introduced a voxel-based pipeline that reconstructs said fields without neural networks by using implicit neural functions to conduct volume rendering.

2.1 Light and Participating Media

From a physical standpoint, light is a form of electromagnetic radiation characterized by a continuous range of wavelengths (λ) and frequencies (ν), with the relationship given by:

$$c = \lambda\nu, \tag{2.1}$$

where c is the speed of light in a vacuum. More importantly, it corresponds to the wavelength range that is visible to the human eye (400 nm to 700 nm) in the band of the electromagnetic spectrum (see Fig. 2.1).

Figure 2.1: Pure spectral colors (Courtesy of Wikipedia).

2.1.1 Interactions between Light and Matter

Visible light interacts with matter in various ways. When a light wave with a single frequency strikes an object, a number of things could happen:

- **Absorption**: energy is absorbed by the medium, converting it to heat.

  Matter is made up of atoms and/or molecules, depending on the substance in question. Ultimately though, both contain electrons which have a natural frequency. When a light wave with the same frequency impinges upon an atom, these electrons get excited and set into vibrational motion (resonance), by effectively absorbing the wave's energy. The Beer-Lambert law describes the decrease in light intensity I as it travels through an absorbing medium over distance z:

  $$I(z) = I_0 e^{-\kappa z}. \tag{2.2}$$

- **Scattering**: energy interacts with the medium changing the direction of light propagation.

  If the wavelength is not changed, i.e., the photon does not undergo a change in energy, the process is called elastic scattering. Conversely, if it does it is called inelastic scattering. Note that for the remainder of this text we assume the latter.

  Scattering can be classified into:

  - *In-scattering*: light scattered towards the observer.
  - *Out-scattering*: light scattered away from the observer.

- **Transmission**: energy passes through the medium without being absorbed or scattered.

  When the aforementioned frequencies do not match up, the electrons instead vibrate for brief periods of time with small amplitudes, reemitting the absorbed energy as a light wave. If the object is transparent the vibrations of the electrons get passed on to those of neighbouring atoms through the bulk of the material.

- **Reflection**: energy bounces off the medium without being absorbed.

  If the object is opaque, the vibrations do not get passed on from atom to atom. Rather the electrons of atoms on the material's surface vibrate and reemit the energy as a reflected light wave.

Participating media are materials that interact with light through emission, absorption and/or scattering (see Fig. 2.2). Examples include fog, clouds, fire, smoke and even clean air to some extent.

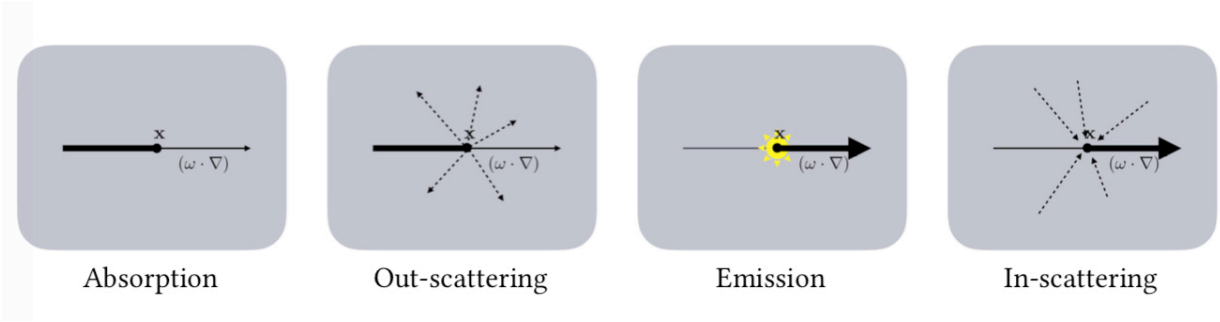


Figure 2.2: Interactions between light and participating media that affect radiance.

2.1.2 Radiative Transfer and Optical Models

Light energy can be described by the measure of radiance I, that is, the amount of radiant flux. It is defined as radiative energy Q per projected unit area A, per solid angle Ω, and per unit of time t:

$$I = \frac{\mathrm{d}Q}{\mathrm{d}A\,\mathrm{d}\Omega\,\cos\theta\,\mathrm{d}t}, \tag{2.3}$$

where Q is the radiant flux, A is the area, Ω is the solid angle, and θ is the angle between the light direction of interest and the surface normal vector.

Radiance is the fundamental quantity for light transport in computer graphics, as it does not change along a light ray in a vacuum. Absorption, emission and scattering all affect radiative energy, so the presence of a participating medium changes radiance levels. The distribution of radiance in volumes is defined by the radiative transfer equation (RTE), defined by Chandrasekhar in 1950. It describes the equilibrium radiance field parametrized by position and direction.

By 1984, Kajiya [12] had already demonstrated the feasibility of simulating light transport in participating media using ray tracing. This seminal work laid the groundwork for understanding and rendering the complex interactions between light and volumetric materials, introducing methods to calculate the attenuation of light within the volume due to absorption and scattering.

Prior to rendering a translucent volume, it is necessary to understand how it interacts with light [13]. For this purpose, optical models are used to describe the behaviour of light within the volume. They are crucial to achieve accurate and realistic visualizations, providing a mathematical framework to simulate light transport.

2.1.3 Volume-Rendering Integral

During the rendering process, the optical model assigns specific optical properties, such as color and opacity, to each voxel. The most widely used model for Direct Volume Rendering is the emission-absorption optical model, which leads to the volume-rendering integral:

$$I(D) = I_0 e^{-\int_{S_0}^{D} k(t)\,dt} + \int_{S_0}^{D} q(s) e^{-\int_{s}^{D} k(t)\,dt}\,ds \tag{2.4}$$

Here, k represents the absorption coefficient, which determines how much light is absorbed by the medium, and q describes the emission coefficient, which accounts for the light emitted by the medium. The integrals compute the cumulative absorption and emission along the entry point S_0 into the volume to the exit point D towards the camera.

- The term $I_0 e^{-\int_{S_0}^{D} k(t)\,dt}$ represents the initial light intensity I_0 attenuated by absorption as it travels through the medium.
- The integral term $\int_{S_0}^{D} q(s) e^{-\int_{s}^{D} k(t)\,dt}\,ds$ represents the contribution of light emitted within the medium, where each point s along the path emits light $q(s)$ that is also attenuated as it travels to the exit point D.

In more complex models, additional are considered to account for light scattering effects such as shadows. These factors further refine the model, enhancing the realism of the rendered image [14].

2.1.4 Compositing schemes

The optical properties are accumulated along each viewing ray to form a 2D projection of the 3D volume data. The accumulated color and opacity are computed according to the discrete volume rendering equation:

$$C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1} (1 - A_j) \tag{2.5}$$

$$A = 1 - \prod_{j=1}^{n} (1 - A_j) \tag{2.6}$$

In these equations, C_i and A_i represent the color and opacity of the voxel at sample i. The opacity A_i evaluates the absorption, while the color C_i approximates the emission, which is weighted by the opacity A_i. Because this equation is a numerical approximation of the continuous optical model, as described by the volume-rendering integral, the sampling rate and the length of the ray segment between sample i and sample $i + 1$ greatly influence the accuracy of the approximation and the quality of the rendering.

Compositing is the process by which samples with the optical properties assigned during data classification and shading are integrated along viewing rays. The theoretical form of the volume-rendering integral [15] is given by:

$$I_\lambda(x, r) = \int_0^L C_\lambda(s) \mu(s) e^{-\int_0^s \mu(t)\,dt}\,ds, \tag{2.7}$$

where $I_\lambda(x, r)$ is the amount of light of wavelength λ coming from ray direction r that is received at location x on the image plane. Here, L is the length of the ray r, $C_\lambda(s)$ is the light of wavelength λ re-

flected and/or emitted at location s in the direction of r, $\mu(s)$ is the density at s, and the exponential is an attenuation function.

Since the volume rendering integral cannot generally be computed analytically, it is often approximated by a Riemann sum of the emitted and absorbed light in intervals i of width Δs:

$$I_\lambda(x,r) \approx \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{2.8}$$

In this approximation, the density μ is substituted by the opacity α, and the exponential function is approximated by a Taylor series. The opacity must be normalized if $\Delta s \neq 1$.

From this approximation, we derive the familiar compositing equations. In front-to-back order, these equations are:

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{2.9}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{2.10}$$

This approach allows for an optimization called early ray termination, which cuts off a ray once the accumulated opacity (or remaining transparency) reaches a threshold where further contributions are negligible, i.e., 0.99 (or 0.01).

In back-to-front order, it is not necessary to keep track of accumulated opacity:

$$C'_i = C_i + (1 - A_i)C'_{i+1} \tag{2.11}$$

This method simplifies the algorithm but does not allow for early ray termination.

Besides compositing, other combining functions can be used. One popular alternative is maximum intensity projection (MIP), which retains the sample with the highest intensity value.

2.2 Data acquisition and representation

The first stage of any visualization pipeline refers to the gathering of data, namely data acquisition. There are multiple methods and devices to generate volumetric data, but they all fall within one of two categories: measured or procedural. The former refers to the result of somehow sampling real objects or natural phenomena. The latter comes from scientifc computer simulations that model a participating medium by a computer simulation or geometric model.

Volume datasets are three-dimensional structures that represent spatially distributed information. More specifically, it can be understood as a continuous scalar field, which can be written as a mapping

$$\varphi : \mathbb{R}^3 \to \mathbb{R}, \tag{2.12}$$

i.e., a function from euclidean 3D space into a single-component real value. This is an ideal conceptual model for mathematical analysis, providing an exact value at any point in the volume that offers infinite precision and resolution.

In a practical context, due to computational limits such as processing power and storage, these continuous fields are approximated by discrete representations, typically defined on a regular grid (see Fig. 2.3). Here, the 3D space is divided into small, finite-sized cells. An individual cell is usually refered to as a voxel [16] [17], and a single value represents the data within each voxel. Note that this discretization introduces a small tolerance, described by the machine epsilon (ϵ).

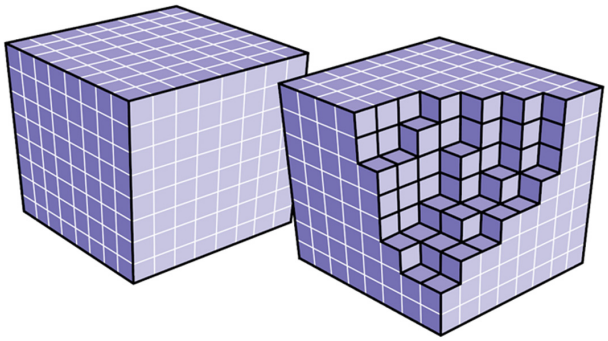


Figure 2.3: Volume dataset representation, given on a discrete uniform grid (Hadwiger et al., 2006).

2.2.1 Measured Data

Most volume datasets contain information from real-world objects or phenomena, captured through some sort of scanning device. This includes data of vastly differently form, size and content, from the physiology of a patient's anatomy to the structure of a mechanical piece (industrial-CT) or some physical property in the micro world.

2.2.1.1 Medical Imaging

One of the earliest scientific fields to adopt volume visualization was medical imaging, which is to this day, one of the primary use cases for volume rendering.

Medical image data is usually represented as a stack of individual images, where each image represents a thin, cross section slice of the scanned body part (volume), composed of individual pixels. These are arranged on a cartesian or uniform 2D grid, where the distance between any two pixels is typically constant regardless of direction. That is, the vertical and horizontal directions have identical distances, the so-called pixel distance. This simplifies the calculation of the actual position by multiplying the respective distance value with the respective pixel index direction. If we assume i indexes the horizontal x position and j indexes the vertical y position, the position of pixel $P_{i,j}$ is easily determined. Fig. 2.4 describes this particular 2D image-space / 3D object-space grid arrangement conversion. Notice how there is no need to index the depth z position, since it is projected onto the 2D grid (flattened image plane).

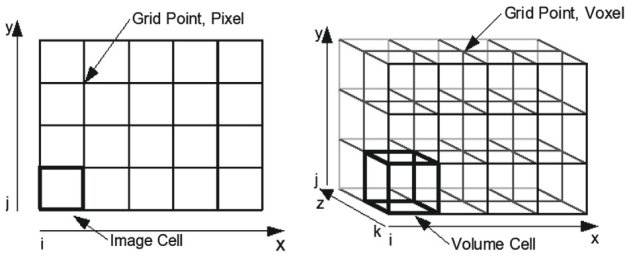


Figure 2.4: Left:2D grid, image pixels on the grid points. Right: 3D grid, individual voxels on the grid points (Courtesy of Dirk Bartz, University of Leipzig).

There exist several specific acquisition methods and devices, known as modalities. In this chapter we shall focus on tomographic imaging modalities, more specifically on CT and MRI data.

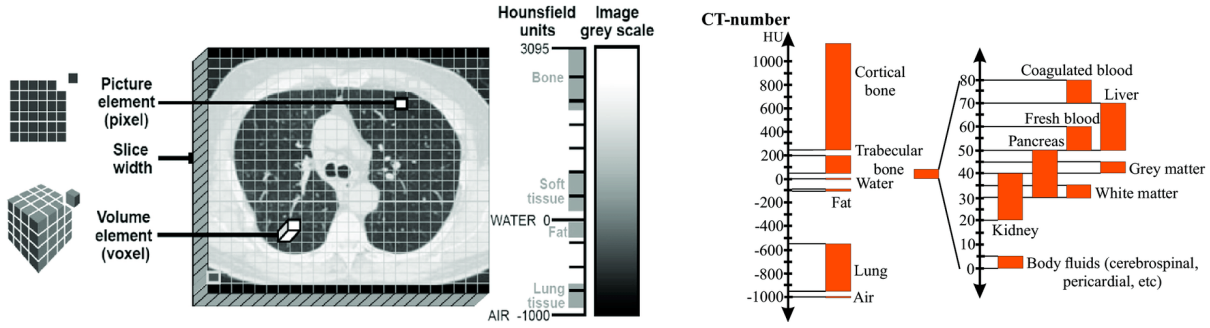
Computed Tomography (CT), including its variants like Positron Emission Tomography (PET) and Single Photon Emission Computed Tomography (SPECT), utilizes X-rays from multiple angles to create detailed 3D images. CT is effective in identifying boundaries between different materials, such as bone and muscle, but struggles with distinguishing similar tissues.

Magnetic Resonance Imaging (MRI) uses strong magnetic fields to align proton magnetic moments in the body. When the field is removed, the protons emit energy as photons, which are detected by the scanner. This method excels at differentiating similar-density tissues, like brain tissue and cerebrospinal fluid, depending on the scanning parameters.

Hybrid PET/CT and PET/MRI scanners highlight the advantages of combining different imaging technologies to address various diagnostic needs. By integrating the strengths of each modality, including advancements like High-field MRI, these scanners provide comprehensive diagnostic information and improved accuracy, enabling clinicians to answer a wide range of clinical questions effectively.

2.2.1.2 Hounsfield scale

The Hounsfield scale, named after Sir Godfrey Hounsfield, is a quantitative measure used in CT scans to describe the radiodensity of materials. It is defined such that the radiodensity of distilled water at standard conditions for pressure and temperature is set to 0 Hounsfield units (HU), while the radiodensity of air under the same conditions is set to -1000 HU. It facilitates the comparison of different tissues' densities by converting the linear attenuation coefficients measured by CT scanners into standardized units (see Fig. 2.5). For instance, bone typically appears with high positive HU values, while fat has negative values, and soft tissues like organs have values near zero. This standardized approach is critical in medical imaging, particularly for precise applications like radiotherapy treatment planning, ensuring consistency and accuracy across different scans and equipment.



(a) Visual depiction of CT-number mappings.

(b) Human body tissues.

Figure 2.5: Hounsfield scale.

2.2.2 Procedurally Simulated Data

Alternatively, datasets may also be procedurally synthesized through computer algorithms, according to some mathematical function. These can be used for testing purposes, when real-world data is not available, or for physical simulations such as computational fluid dynamics or geological subsurface layers.

It is even possible to model fuzzy, undefined boundary objects. This is especially useful for depicting pockets of gas like gaseous atmospheric phenomena. In fact, several state-of-the-art videogames feature volumetric clouds, fog or lighting. While some of these effects can be achieved/-faked by traditional polygonal geometry, their volume rendered counterpart usually ends up being more realistic (Schpok et al., 2003) [18].

2.3 Sampling and Filtering

As it has already been stated, in practical applications volume data takes a discretized form, such as a uniform or tetrahedral grid. Therefore, volume rendering is performed using numerical approximation, i.e., approximating integrals with Riemann sums.

The integral of a function along a ray can be approximated by summing up the function's values at discrete sample points, weighted by the interval between samples (the step size).

$$I \approx \sum_{i=1}^{N} f(x_i)\Delta x \tag{2.13}$$

Here, I is the integral, $f(x_i)$ is the value of the volume data at the i-th sample point, and Δx is the distance between sample points.

2.3.1 Nyquist-Shannon Theorem

The Nyquist-Shannon sampling theorem, which is a fundamental principle in the field of digital signal processing, can be stated mathematically as follows:

$$f_s \geq 2B \tag{2.14}$$

Here, f_s is the sampling frequency (the number of samples per second), and B is the bandwidth of the signal (the highest frequency contained in the signal).

This theorem can also be expressed in terms of the signal reconstruction:

$$x(t) = \sum_{n=-\infty}^{\infty} x(nT)\,\text{sinc}\left(\frac{t-nT}{T}\right) \tag{2.15}$$

In this equation:

- $x(t)$ is the original continuous signal.
- $x(nT)$ are the sampled values of the signal at intervals of $T = \frac{1}{f_s}$.
- $\text{sinc}(x)$ is known as the sampling function, defined as $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$.

2.3.2 Signal Reconstruction

Discretization leads to the issue of reconstructing the function φ of (2.12) on all points in the 3D domain. Data values are generally available as samples on a grid. When data is needed at non-grid points, their values must somehow be estimated from the surrounding grid points. To accurately do so, interpolation is necessary.

Uniform grids facilitate tensor-product reconstructions, which are particularly useful for implementing efficient and effective interpolation methods. One common approach is the 3D tensor-product reconstruction filter, especially for linear interpolation. This method interpolates data by considering the contribution of adjacent grid points along each dimension. Specifically, trilinear interpolation (see Fig. 2.6), a form of tensor-product interpolation, performs linear interpolation along each axis sequentially.

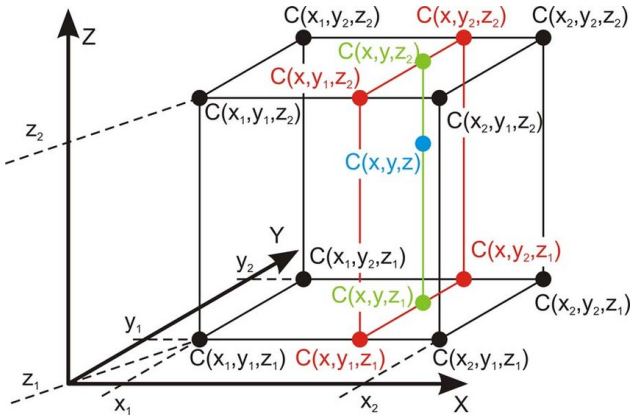


Figure 2.6: Trilinear interpolation depiction in 3D.

Other interpolation methods like cubic splines and gaussian variants offer improved accuracy, but they do so at the cost of higher computational complexity and the lack of specialized hardware support. This is why modern GPU implementations of Direct Volume Rendering typically use

hardware-supported trilinear interpolation. It strikes a good balance between quality and complexity. Additionally, using weights such as the bilateral kernel can further enhance results by accounting for both spatial and radiometric distances; thus improving the reconstruction's adaptability to varying data features.

2.4 Volume Visualization Pipeline

So far, we have discussed data acquisition and data filtering and mentioned seminal methods that project the 3D scalar data onto a flat, 2D image plane (see Fig. 2.7).

Figure 2.7: Volume Visualization Pipeline.

A very simple and naive way to inspect the data is per axial slices, mapping scalars to intensity values to form a grayscale image. However, in most cases we want to perform this so-called projection, as it shows inherent spatial structures that enhance understanding.

2.4.1 Indirect and direct volume visualization

There are two main approaches to rendering a volume:

- **Indirect:** the volume is reduced to an intermediate surface representation, i.e., a standard polygonal mesh, which can then be drawn onto the screen by means of rasterization (see Fig. 2.8).

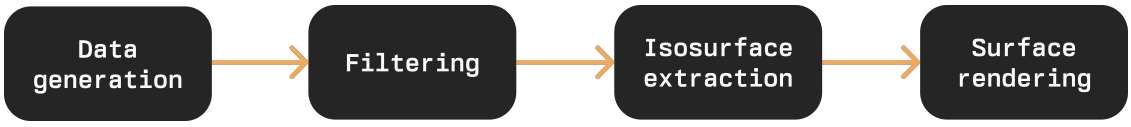


Figure 2.8: Indirect volume visualization.

- **Direct:** the volume gets evaluated for an optical model. It is considered as an absorptive, light-emitting medium. The visual impression is then simulated according to the laws of physics for each pixel. This is known as direct volume rendering (DVR), and while there are different computational solvers the most common is volume ray casting / ray marching (see Fig. 2.9).

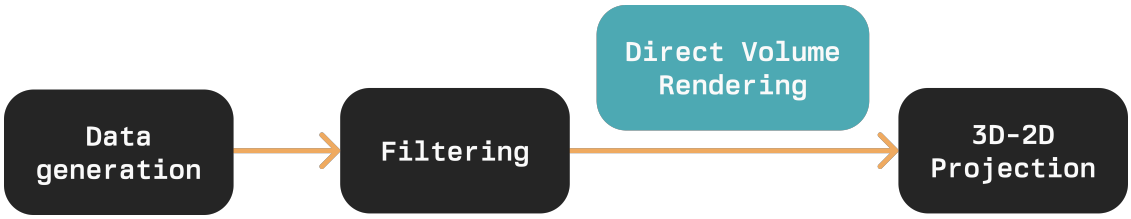


Figure 2.9: Direct volume visualization.

2.4.2 Object-order, image-order and domain-based techniques

Volume rendering techniques can be classified in the following categories:

- **Object-order:** forward mapping scheme, where each voxel unit is mapped onto the image plane. Example: splatting.
- **Image-order:** backward mapping scheme, where rays are cast from each pixel unit in the image plane through the volume data to determine the final color value.
- **Domain-based:** the spatial volume data is transformed into some other domain (frequency, wavelet, etc.) from which the projection is generated.

As a side note, hybrid rendering methods, employing several of the above, are also possible.

Chapter 3

Problem statement and requirements

In this text we shall explore the development of a small desktop application capable of loading in volumetric datasets that get passed on to the graphics processing unit to offload rendering them efficiently in a virtual environment. This entails performing volumetric ray-casting and using transfer functions to map RGBa tuples to radiodensity values within the volume. These will then get composited to synthesize the final image. It is often desirable to be able to interact with the displayed volume, so basic light and camera simulation is also needed.

This type of software is especially useful in the biomedical field, being common among radiologist and other clinicians. Thus, the emphasis will be on visualization of CT and MRI scans.

In order to have a good user experience some considerations need to be made (beyond UI design). Low input latency and high frame rates are crucial for attaining a snappy, responsive application.

3.1 Use cases

Finally, after consulting with some nurses and a doctor, the following use cases have been defined (see Fig. 3.1 to 3.6) as essential functionalities to enhance either visualization, diagnosis or planning procedures:

Use case	Load in a volume.
Description	Deserialization of a volume dataset, which may be encoded in some raw binary format, open standards or a set of images. It will be passed on to the GPU using 3D textures.
Workflow	Toggle for predefined test files or call to action button that prompts a file explorer to select the dataset file from disk.
Precondition	None. If a volume is already loaded, upon opening a new one they shall be swapped.
Postcondition	Display the volume in the viewport using the default rendering mode (transfer function).

Figure 3.1: First use case.

Use case	Rotation about a point & zooming towards it.
Description	In order to explore the volume in a 3D-space, it is necessary to implement a look-at camera with rotation and zooming capabilities.
Workflow	Use special keys and mouse clicks/movements to zoom in and rotate around the volume.
Precondition	Load in a volume.
Postcondition	Camera (eye) matrix transformation.

Figure 3.2: Second use case .

Use case	Rendering mode: maximum intensity projection (MIP).
Description	MIP projects the voxels with maximum intensity (along each ray) onto the image plane. It is often used to detect lung cancer nodules (CT), as well as other high intensity structures with angiography (MRI).
Workflow	The volume gets assigned the “MIP” fragment shader and rendered into the default framebuffer.
Precondition	Load in a volume.
Postcondition	Display the volume in the viewport.

Figure 3.3: Third use case.

Use case	Rendering mode: average intensity projection (AIP).
Description	AIP projects the averaged intensity values. This type of visualization is known as pseudo X-Ray.
Workflow	The volume gets assigned the “contour” fragment shader and rendered into the default framebuffer.
Precondition	Load in a volume.
Postcondition	Display the volume in the viewport.

Figure 3.4: Fourth use case.

Use case	Optimization.
Description	Implement early-ray termination, to avoid indexing the ray marching loop at unnecessary sample locations.
Workflow	Upon surpassing a high opacity threshold, return and stop moving along the ray.
Precondition	Load in a volume.
Postcondition	Break out of the ray marching loop.

Figure 3.5: Fifth use case.

Use case	Rendering mode: simplified transfer function.
Description	Simple transfer function which maps an opaque color to samples of a given intensity range, and a semi-transparent one to those above or below it.
Workflow	The volume gets assigned the “simplified TF” fragment shader and rendered into the default framebuffer.
Precondition	Load in a volume.
Postcondition	Display the volume in the viewport.

Figure 3.6: Sixth use case.

Chapter 4

Design

4.1 UX/UI Prototyping

In this section, we present the prototyping process for the user experience (UX) and user interface (UI) of the desktop application. First of all, a list of parameters was extracted from the predefined use cases:

- **Rendering mode:** selectable for the different types of visualization.
- **Step size:** ray marching increment distance; it defines how far we should march along the viewing ray between iterations.
- **Density factor:** global scalar multiplier that gets applied to all voxels.
- **Mid value:** density value of interest. It only affects the transfer function mode.
- **Range:** interval around the mid value to use for data classification. It only affects the transfer function mode.
- **Tint color:** hue of the volume.

4.1.1 High-fidelity wireframe

The UI consists of a floating window dedicated to adjusting volume rendering parameters. It can be clicked on and dragged around so as not to have it in the way of the volume. Note that the application window acts as a canvas, meaning that the synthesized image output gets displayed on it in real-time.

This design prioritizes simplicity and functionality, providing users with an immersive, non-intrusive experience that focuses on the visualization. Below is a UI mockup created in Figma (see Fig. 4.1), demonstrating the overall layout and functionality envisioned for the application:

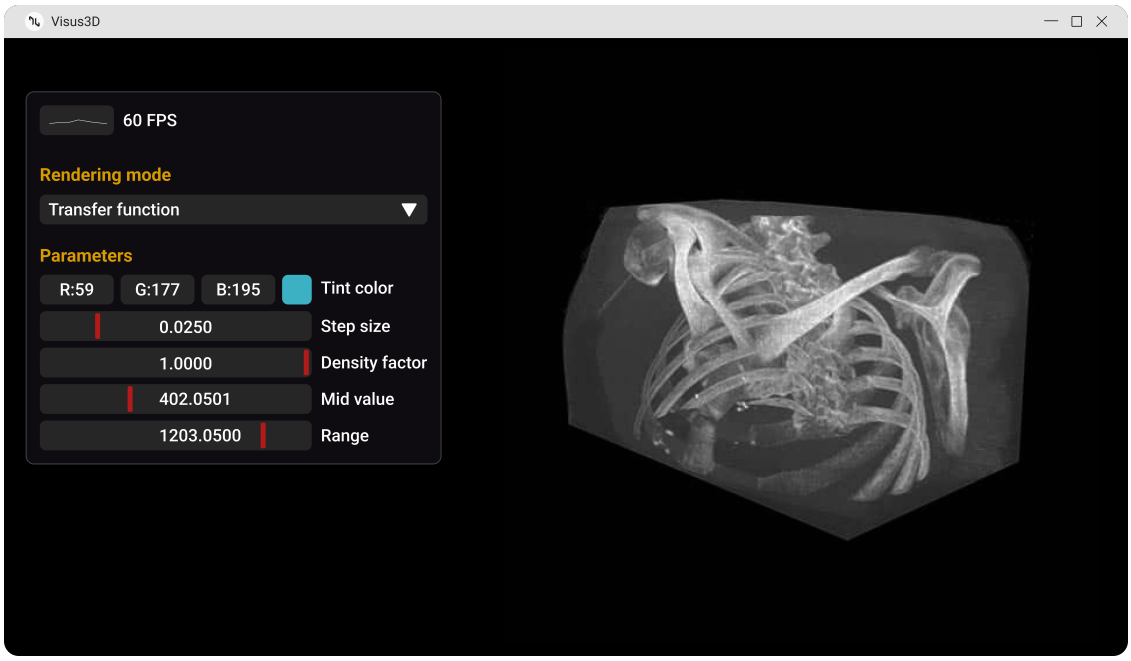


Figure 4.1: UI mockup.

4.2 Software Development

4.2.1 Methodology

Feature-Driven Development (FDD) was selected as the development framework. FDD is a model-driven short-iteration methodology that emphasizes building and delivering features incrementally, in a structured and predictable manner.

As a solo developer, this approach proves particularly advantageous given the need to efficiently manage a backlog of features without the need to rely on daily meetings for communication, as is usually the case for other agile frameworks like SCRUM.

4.2.2 Software architecture

4.2.2.1 Data and logic decoupling

In modern software development modularity, scalability, and maintainability are paramount. Achieving these requires a thoughtful design that emphasizes both logic and data decoupling. This allows for different parts of an application to be developed, updated, and tested independently, reducing complexity and improving adaptability.

While practices like maintaining global state are typically discouraged in other types of software development, they are more often than not used in graphics applications, such as games. For the most part, some level of coupling is unavoidable, as you are bound to the design choices of the platform's SDK or third-party libraries that abstract this away.

Despite this, it is essential to balance these practices with principles that promote modularity and separation of concerns, ensuring that the overall architecture remains robust and adaptable.

We settled on a design that is structured around four layers:

- **Core:** windowing and input event system. It provides the foundation to interact with the operating system's SDK.
- **Graphics:** a set of OpenGL constructs for all computer graphics needs, abstracting and partially automating the rendering process.
- **UI:** graphical user interface creation and logic.
- **App:** actions for the application's logic. It communicates with the other layers.

4.2.2.2 Project structure

A well-structured project can act as a roadmap, guiding developers efficiently to locate files and components. It is key for organization, clarity and consistency. Fig. 4.2 shows the tree directory structure for the project.

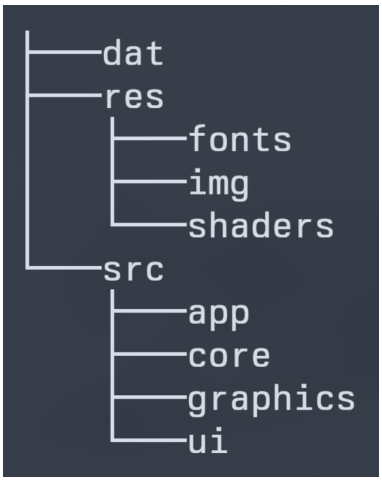


Figure 4.2: Tree project structure.

- **dat:** volume datasets.
- **res:** miscellaneous files, from fonts and images to shader source code.
- **src:** main source code.

4.2.2.3 Class diagram

Next, we need to define the entities and their relationship between one another (see Fig. 4.3). It is important to note that this does not strictly follow UML, as our code base somewhat blends object-oriented and procedural styles, by using a combination of classes, runtime polymorphism and free functions within namespaces.

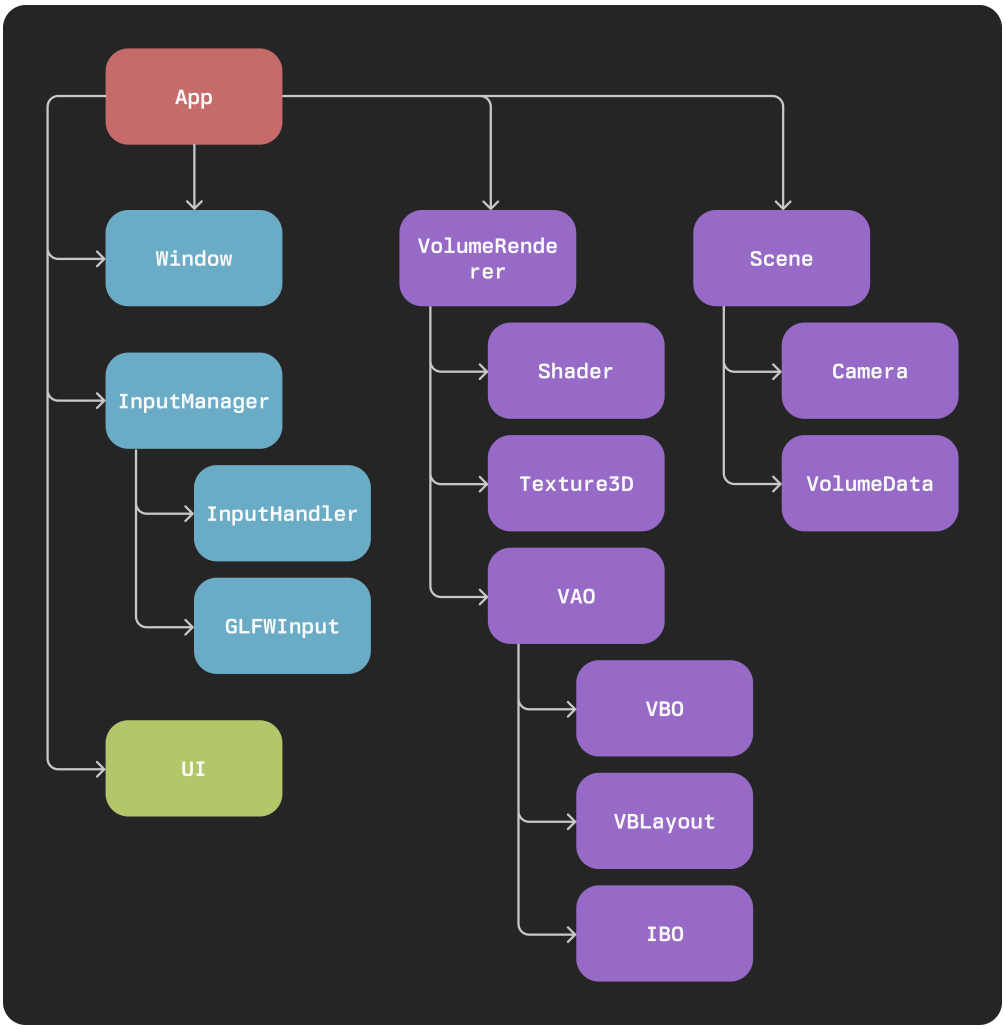


Figure 4.3: Class diagram.

All classes have been color coded according to the layer they belong to:

- **Red:** app layer.
  - *App:* application instance.
- **Green:** UI layer.
  - *UI:* creation and management of GUI.
- **Blue:** core layer.
  - *Window:* window creation and management.
  - *InputManager:* bind actions to input handlers, and input handlers to backend callbacks (triggered by hardware events).
  - *InputHandler:* polymorphic API to handle input events. It uses `std::function` (a C++ wrapper for callable objects) and lambdas, making it possible to swap input backends even at runtime.

  - *GLFWInput:* bind GLFW callbacks and redirect them to the application input handler.

- **Purple:** graphics layer.

  - *VAO, VBO, VBLayout, IBO:* vertex array object, vertex buffer object and its layout (stride attributes), index buffer object. These are necessary for the creation of a fullscreen quad, so that we have a pixel shader to perform image-order direct volume rendering.
  - *Shader:* shader program to perform volume rendering. It attaches both the vertex shader source and the fragment shader source.
  - *Texture3D:* volumetric data gets encoded in a three-dimensional scalar field. This maps nicely to a 3D texture which can be bound to the shader program and passed on to the GPU.
  - *Scene:* definition of all entities in the environment.
  - *Camera:* orbit camera creation and manipulation (eye transformation).
  - *VolumeData:* NIfTI file deserialization and processing.
  - *VolumeRenderer:* renderer, it connects to the app instance via a user data pointer to retrieve whatever data it needs to perform direct volume rendering.

4.2.2.4 Entry point

The program's entry point interacts solely with the app layer, creating an instance of an app from which to invoke the execution of a frame loop, i.e., the main loop (see Fig. 4.4). The loop itself orchestrates the creation and management of the window and all other resources, running indefinitely over many frames per second (FPS).

```
#include "app/App.hpp"

int main()
{
    visus::App visus3D;
    visus3D.frameloop();

    return 0;
}
```

Figure 4.4: Entry point (main.cpp).

Chapter 5

Implementation

5.1 Tools and technologies

Traditionally, most graphics applications are written in rather low-level programming languages such as C and C++, since they offer fine-grained control over hardware resources. This is crucial in graphics programming, where memory management, cache locality and the number of processor instructions can significantly impact rendering speed. Moreover, these languages support advanced features such as pointer arithmetic and manual memory management, which are valuable in optimizing graphics code.

We have chosen C++ 17 as the primary language because it provides modern features which allow for very flexible high-performance systems with some added code safety. OpenGL has been picked as the graphics API for its cross-platform support and relative ease of use, compared to other much more verbose APIs, such as Vulkan and DirectX 12. Therefore, GLSL is also used for shader code.

Here is a list outlining the entire development environment:

- **Version control system:** GIT.
- **Text editor:** GNU Emacs.
- **Language Server Protocol:** clangd.
- **Compiler:** MSVC.
- **Build system:** CMake.
- **Package manager:** vcpkg.
- **Code formatter:** clang-format.
- **Code linter (static code analysis):** clang-tidy.
- **Graphical debugger:** RAD Debugger.

- **Graphics debugger:** RenderDoc.

And, the dependencies:

- **GLFW:** multi-platform desktop windowing, graphics context and input event management.
- **GLM:** graphics math, especially useful for matrix algebra structures and operations. It is based on GLSL.
- **glbinding:** C++ OpenGL function loader.
- **nifticlib:** C libraries to add NIFTI support, i.e., I/O for binary files storing medical image data.
- **Dear ImGui:** bloat-free immediate mode graphical user interface library.
- **stb:** header-only public domain libraries for C/C++. Particularly, *stb_image.h*, to load PNG image files.

5.2 Core layer

5.2.1 Windowing

The window creation process involves a bit more than simply opening up a window. First of all, we initialize GLFW and set an error callback to get error messages if anything were to fail. Secondly, we initialize a graphics context with OpenGL. By default GLFW may create a context with any version, yet we may specify a minimum required version. If it is not supported by the current platform, both context and window creation will fail. We also enable multisample anti-aliasing (MSAA) and double buffering to avoid flickering. This way we always render to the back buffer and display the front buffer on screen. At the end of our render call we swap the buffers, such that the back buffer becomes the front buffer and viceversa. Lastly, we enable VSync (synchronizing the display's refresh rate with the graphics card's frame rate to prevent screen tearing), set the window title and the window icon with a custom logotype.

Additionally, we store the handle to the window in the user data pointer provided by GLFW, so that we may access it from anywhere.

5.2.2 Input system

When it comes to the input system, we are faced with a choice: input polling or event-based input. While both options have their pros and cons, we have gone with the latter as it proves to be a bit more efficient; only processing input when events happen.

Employing modern C++ concepts such as lambdas and `std::function` (a wrapper for callable objects) we achieve a clean and flexible API that relies on runtime polymorphism. This design makes it trivial to swap backends. For example: replacing GLFW with SDL for input. It also enables the definition of actions separate from the callbacks of hardware triggers. Below you may find some illustrative code extracts.

```
// Frontend handlers
struct InputEventHandlers
{
    std::function<void(KeyEvent e)> onKeyPressed;
    std::function<void(MButtonPressEvent e)> onMButtonPressed;
    std::function<void(MCursorMoveEvent e)> onCursorMoved;
    std::function<void(MScrollEvent e)> onScrolled;
};

// Backend handlers
struct GLFWInputHandlers
{
    // Keyboard events
    std::function<void(const int32_t key, const int32_t scanCode, const
    int32_t action,
                       const int32_t mods)>
        onKeyPress;

    // Mouse events
    std::function<void(const int32_t button, const int32_t action, const
    int32_t mods)>
        onMouseButtonPress;
    std::function<void(const double xPos, const double yPos)> onCursorMove;
    std::function<void(const double xOffset, const double yOffset)> onScroll;
};

inline GLFWInputHandlers glfwInput;
inline InputEventHandlers input;
```

Listing 5.1: Event handlers

```
input.onCursorMoved = [hwnd](MCursorMoveEvent e)
{
    // NOTE(abi): we don't want to modify the 3D view while using the GUI.
    if (App* app = static_cast<App*>(glfwGetWindowUserPointer(hwnd));
        app && !ImGui::GetIO().WantCaptureMouse)
    {
        app->getScene()->getCamera()->onCursorMove(e.xPos, e.yPos);
    }
};
```

Listing 5.2: Binding an action to an input handler

5.3 Graphics layer

5.3.1 Arcball camera

To be able to explore the volume we need a camera. Now, the graphics pipeline is not familiar with the notion of a camera per se, but we can simulate one through the eye/view matrix. Effectively, we move all objects in the scene in the reverse direction, giving the illusion of moving.

A simple and intuitive way to explore the volume is to orbit around it. Since it is the only object in the scene this makes the most sense, such that the camera is always looking at the volume. We could implement tumbling, which is what many 3D software suites do. However, most implementations use a virtual sphere whose rotation is not confined to a specific axis, making it nearly impossible to get precise rotations around global coordinate axes. A nicer alternative is the so-called arcball camera, presented by Ken Shoemake in 1992 [19].

The arcball camera uses an invisible virtual sphere centered around a pivot or look-at point to map 2D mouse movements to 3D rotations. The sphere is projected onto the screen as a unit circle, but only the hemisphere facing the user is used for input, as it represents the part of the sphere that can be rotated with a single mouse motion. Upon mouse movement, it creates vectors on the hemisphere (see code snippet below) that are then used to calculate rotation around the pivot point (see Fig. 5.1). This system has endured the test of time and proved intuitive by the HCI community.

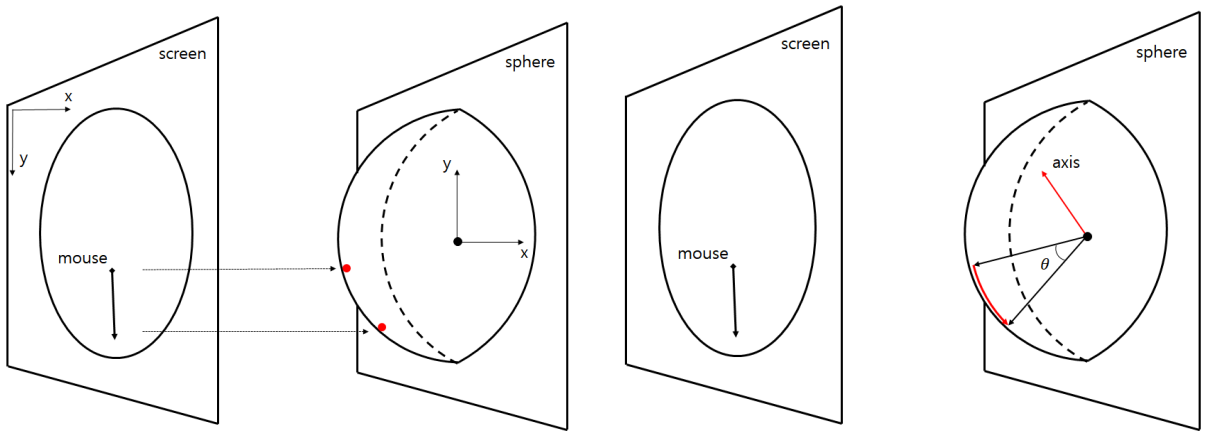


Figure 5.1: Screen to arcball space.

```
glm::vec3 Camera::screenToArcball(glm::ivec2 point) const
{
    // Screen -> NDC
    // NOTE(abi): in screen space the origin is at the top-left corner
    // (Y+ down), so we need to flip it.
    glm::vec2 pointNDC{
        2.f * static_cast<float>(point.x) / static_cast<float>(_viewport[0])
    - 1.f,
        -2.f * static_cast<float>(point.y) / static_cast<float>(_viewport[1])
    + 1.f};
```

```
9
10      // NDC -> Arcball
11      // Project onto the arcball's hemisphere
12      glm::vec3 pointArcball{pointNDC, 0.f};
13
14      // Within unit circle (euclidean distance)
15      if (float lengthSq = glm::dot(pointNDC, pointNDC); lengthSq <= 1.f)
16      {
17          pointArcball.z = sqrt(1.f - lengthSq);
18          return pointArcball;
19      }
20
21      // Outside (circle boundary -> sphere rim)
22      pointArcball = glm::normalize(pointArcball);
23      return pointArcball;
24  }
```

Listing 5.3: Screen space point to arcball space vector implementation.

To rotate we have several options: a rotation matrix, quaternions or even rotors (from Grassmann algebra). Likewise, to zoom in and out we could use different affine transformations, to either change the field of view or translate the camera (eye) forward or backward.

```
1  void Camera::rotate(const double& x, const double& y)
2  {
3      glm::vec2 prev{_cursor.x, _cursor.y};
4      glm::vec2 curr{x, y};
5
6      if (prev != curr)
7      {
8          // Map points to hemisphere and computer vectors
9          glm::vec3 a = screenToArcball(prev);
10         glm::vec3 b = screenToArcball(curr);
11
12         // Angle and rotation axis
13         glm::vec3 axis = glm::cross(a, b);
14         float angle = glm::acos(glm::max(-1.f,
15                                 glm::min(1.f, glm::dot(a, b))));
16
17         // Transform back the axis and rotate
18         glm::vec4 transformedAxis = glm::inverse(_viewMat) *
19                                     glm::vec4(axis, 0.0);
20         _viewMat = glm::rotate(_viewMat, angle, glm::vec3(transformedAxis));
21     }
22 }
```

Listing 5.4: Camera rotation.

```
// NOTE(abi): instead of translating the camera we modify the FoV
// (lens-like zoom).
void Camera::zoom(const float factor)
{
    _fov = glm::clamp(_fov - (factor * ZOOM_SENSITIVITY), 1.f, 90.f);
    _projectionMat =
        glm::perspective(glm::radians(_fov), _viewport[0] / _viewport[1],
    _near, _far);
}
```

Listing 5.5: Camera zoom.

5.3.2 Data deserialization

We focus on deserializing medical imaging data using the NIfTI-1 format, a widely used standard in neuroimaging and fMRI in particular. Unlike raw data formats that lack inherent information, NIfTI files store both binary image data (voxel values) and embedded metadata (dimensions, voxel size, data type, etc.). It strikes the right balance between simplicity and standarization, making it a preferred choice over formats like DICOM and PVM.

We employ *nifticlib*, a C library tailored for reading and writing NIfTI files. It facilitates access to both the header metadata and the actual image data.

5.3.3 CPU and GPU interfacing

CPUs excel at sequential tasks and general-purpose computing, whereas GPUs shine in highly parallel and data-intensive applications, such as rendering and machine learning. For this very reason, we carry out volume rendering in the GPU, through the programmable graphics pipeline (see Fig. 5.2) [20].

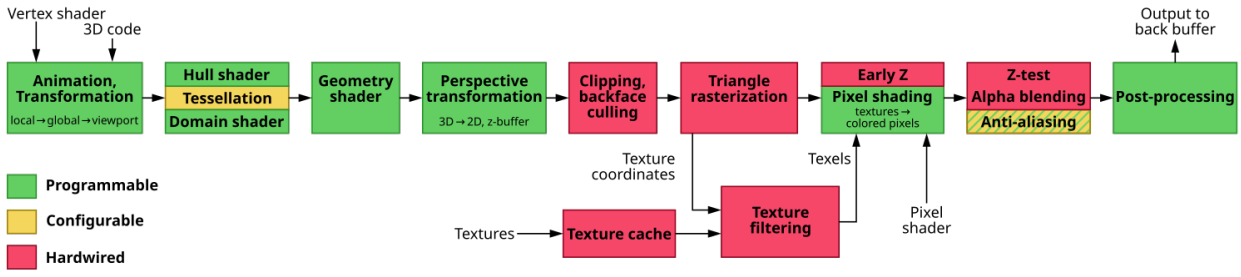


Figure 5.2: Programmable graphics pipeline (Courtesy of Wikipedia).

In order to forward data to the GPU we use flat uniforms. These are variables set by the application that get sent to the GPU alongside the draw call. Their values are stored in the program shader, such that they are accessible through different stages of the graphics pipeline.

5.3.4 Volume ray marching

Having laid out all the groundwork, it is finally time to render a volume. To do so, we shall take a closer look at image-based direct volume rendering, more specifically volume ray casting / ray marching. A good starting point for this topic is the book "Real-Time Volume Graphics", by Klaus Engel et al. [14]. Current implementations may differ greatly from the ones shown in it, due to hardware development, but it demonstrates foundational techniques that have stood the test of time and remain relevant.

Before discussing the overall algorithm, a few technical remarks are appropriate to provide some context:

- Given that a volumetric dataset is encoded in a finite 3D scalar field, it maps nicely to a 3D texture that spans from coordinates $(0, 0, 0)$ to $(1, 1, 1)$. Querying this texture at any point within its bounds will yield the radiodensity value of the scanned object at that relative location.
- A bounding primitive encasing the volume is defined, so that there are clear entry and exit points for any intersecting ray. These delimit a ray segment, which is key because, unlike ray tracing, ray casting does not compute intersection points.

Here is a high-level overview of a classical DVR pipeline:

1. **Data traversal:** march through the volume in fixed incremental steps.
2. **Interpolation:** at each sampling location, a value is reconstructed from the voxel grid by an interpolation scheme (nearest neighbour, trilinear, b-spline filtering, ...).
3. **Normal estimation:** compute the gradient, i.e., the rate of change in data values. Common kernels include intermediate difference, central difference and sobel operator. This is needed for shading (if applicable [1]).
4. **Data classification:** a transfer function T maps data values R to colors C; $T : \mathbb{R} \to \mathbb{C}$. Typically, the color is an RGB triple of either floating-point, $[0, 1]$, or unsigned char, $[0, 255]$, values. There are two types:
   - *Pre-classification:* apply T to sample values, then interpolate colors.
   - *Post-classification:* interpolate data first, then apply T.
5. **Shading:** apply illumination models, e.g., Phong reflection model (if applicable [1]).
6. **Compositing:** combine samples (back-to-front or front-to-back).
7. **Image display:** blit the composited image to the screen [21].

In short, by precomputing entry and exit points we only render ray segments that fall within the volume. For each pixel we cast a ray that starts at the entry point, traverses the volume in steps and ends in the exit point. At each step, we sample the view direction and convert interpolated

[1] To simulate lighting we can approximate normal vectors and apply a shading model.

voxel data values into color values. Lastly, we add up all color contributions with alpha blending to get the output color and opacity of the pixel. Figure 5.3 illustrates this process.

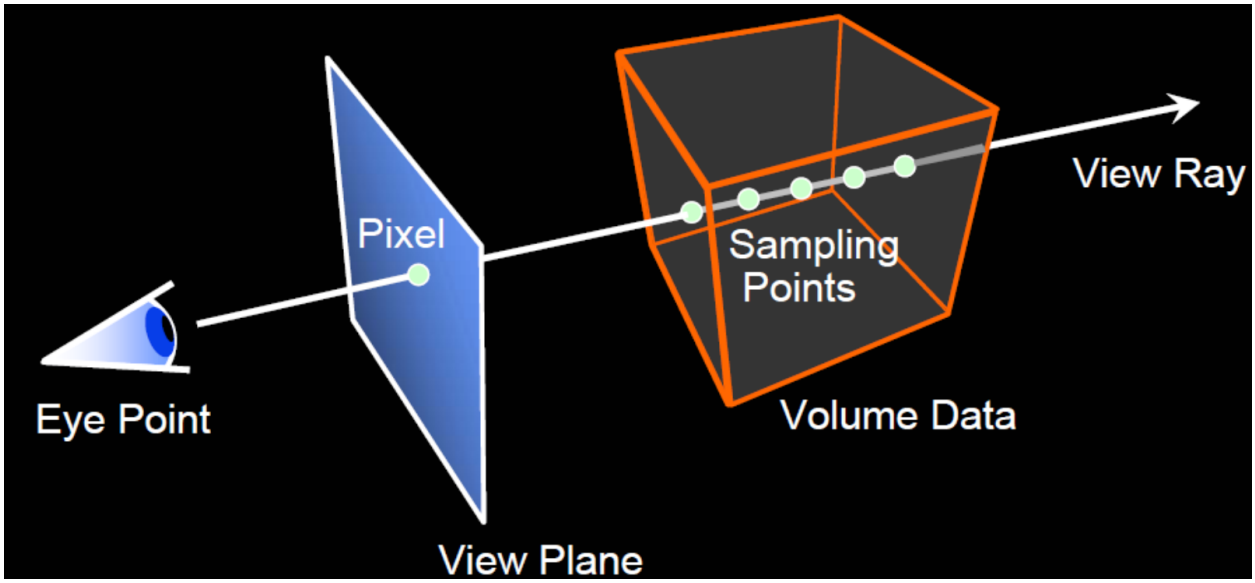


Figure 5.3: Volume ray casting.

GPU volume ray casting can be performed with either two render passes or a single one. Two-pass raycasting is easier to implement as you can set up a unit cube with face culling enabled and rasterize front and back textures (which determine entry and exit points). By substracting these we get the ray direction (see Fig. 5.4).

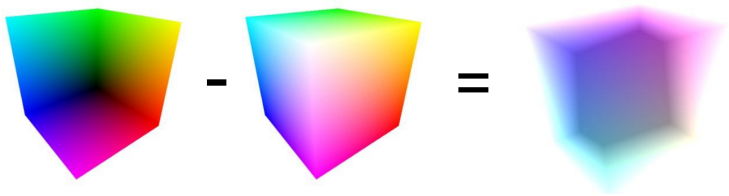


Figure 5.4: Back, front and direction vector textures (from left to right).

Alternatively, volume ray casting can be done in a single pass, shaving off a draw call. There are different approaches, but we have decided to create a naive implicit bounding box within the fragment shader itself, such that we do not need to rasterize the textures from Fig. 5.4.

Here is an in-depth description of our single-pass volume ray casting algorithm:

1. We draw a fullscreen quad (two triangles), by setting up a vertex array object with its corresponding vertex and index buffers. We also set a 3D texture with the volume data and pass it along as a uniform.

2. Vertices get processed in the vertex shader, where we simply pass on the positions to the fragment shader. Note that vertex coordinates get interpolated across the surface of each

triangle using barycentric coordinates. This way we may retrieve the position of any fragment/pixel on the screen.

3. Within the fragment shader we set up the viewing ray (origin and direction) and step through the volume incrementally. We define an implicit viewing volume in front of the camera and multiply it by the inverse model view projection transform [22]. This serves as our bounding box, shooting rays from the near plane and terminating them when they are past the far plane.

```
1    // Ray setup
2    vec4 near = inverseModelViewProjectionMatrix * vec4(fragPosition,
    -1.0, 1.0);
3    near /= near.w;
4
5    vec4 far = inverseModelViewProjectionMatrix * vec4(fragPosition, 1.0,
     1.0);
6    far /= far.w;
7
8    vec3 rayOrigin = near.xyz;
9    vec3 rayDirection = normalize((far - near).xyz);
10
11   // Naive ray segment
12   vec3 enterPoint = rayOrigin;
13   vec3 exitPoint = rayOrigin + (rayDirection * 5);
14
```

Listing 5.6: Implicit bounding volume.

4. At each sample point, we query the Sampler3D holding the texture unit with the voxel data.

5. Depending on where the sampled value falls within a custom range we apply a certain color and transparency value. This is a simple 1D transfer function. We set a threshold for very low values to avoid picking up the air's density.

6. At the end, we blend together the colors at every step along the viewing ray, to produce the final color of the current fragment. If a pixel's opacity gets saturated we terminate the ray marching loop early. This is known as early ray-termination, a simple but effective optimization. Note that for this to work, we must use front-to-back compositing.

```
1    vec4 dvr(vec3 rayOrigin, vec3 rayDirection, vec3 enterPoint, vec3
    exitPoint)
2    {
3        float maxDistance = length(exitPoint - enterPoint);
4        vec3 currentPosition;
5        float totalDistanceTraveled = 0.0;
6
7        vec3 currentColor = vec3(0.0);
8        float currentAlpha = 0.0;
9
10       vec3 accumulatedColor = vec3(0.0);
```

```
11        float accumulatedAlpha = 0.0;
12
13        // Ray marching loop
14        while (totalDistanceTraveled < maxDistance)
15        {
16            currentPosition = enterPoint + totalDistanceTraveled *
    rayDirection;
17            float volumeTextureValue = texture(volumeTexture,
    currentPosition).r;
18
19            // Map radiodensity to RGBa tuple
20            vec4 currentSample = transferFunction(volumeTextureValue);
21            currentColor = currentSample.xyz;
22            currentAlpha = currentSample.w;
23
24            // Compositing
25            float colorAlpha = currentAlpha - (currentAlpha *
    accumulatedAlpha);
26            colorAlpha *= densityMultiplier;
27            accumulatedAlpha += colorAlpha;
28            accumulatedColor += currentColor * colorAlpha;
29
30            // Early ray-termination
31            if (accumulatedAlpha > 0.99)
32            {
33                break;
34            }
35
36            // March along
37            totalDistanceTraveled += marchDistance;
38        }
39
40        return vec4(tfHue * accumulatedColor, accumulatedAlpha);
41    }
42
```

Listing 5.7: Direct volume rendering function.

This fragment shader gets executed in parallel for every pixel on the screen, therefore synthesizing the whole picture.

5.3.4.1 Visualization modes

It is possible to get various other visualization effects, by applying different projection methods. That is, instead of using a transfer function, we return some statistical value of interest: maximum, median, mean, etc.

Here is a list of the rendering modes our application supports:

- **Transfer function:** it applies a 1D transfer function as described in the previous section. It offers the most customization; we can segment isovalues by filtering out a specific range of densities.
- **Maximum Intensity Projection (MIP):** it returns the maximum density value along each viewing ray. This is of special interest for medical image analysis. It is used to enhance the visibility of vascular structures in Magnetic Resonance Angiography (MRA), as well as detecting pulmonary nodules in chest radiographs; a common indicative of pulmonary cancer.
- **Average Intensity Projection (AIP):** it returns the mean density value along the viewing ray. This projection is known as ”pseudo X-Ray”, as it closely resembles one, making it useful to inspect bone tissue.

5.4 Challenges

In this section we shall comment some of the hurdles encountered during the implementation phase and their solutions:

- **Stack overflow:** step debugging in RAD Debugger to figure out the problem. The fix was to use some heap allocations using smart pointers.
- **Unitialized lambdas:** again step debugging came in useful. All that was needed was to initialize them.
- **Vertex window order:** this took a while to figure out. RenderDoc came in useful to inspect the draw call, particularly the vertex and index buffer arrays. The problem was an incorrect layout setup for the vertex stride.
- **Blank screen:** the OpenGL state was being reset at the end of the render loop, clearing out the color buffer of the default framebuffer.
- **Unexpected fragment outputs:** shader debugging is hard and tedious. This took a lot of back and forth: reviewing my notes, using RenderDoc, looking through Stack Overflow threads and testing out solutions.

5.5 Repository

For further insights into the implementation, please refer to the project’s code repository on GitHub: https://github.com/abidanBrito/visus.

Chapter 6

Results and evaluation

6.1 Screenshots

In this section, we present various screenshots of the final application (see Fig. 6.1), showcasing its features and functionalities. These images provide a visual overview of what is possible (choosing between several visualization modes, custom 1D transfer function, color tinting and tweaking global density and increment size for the ray marching loop).

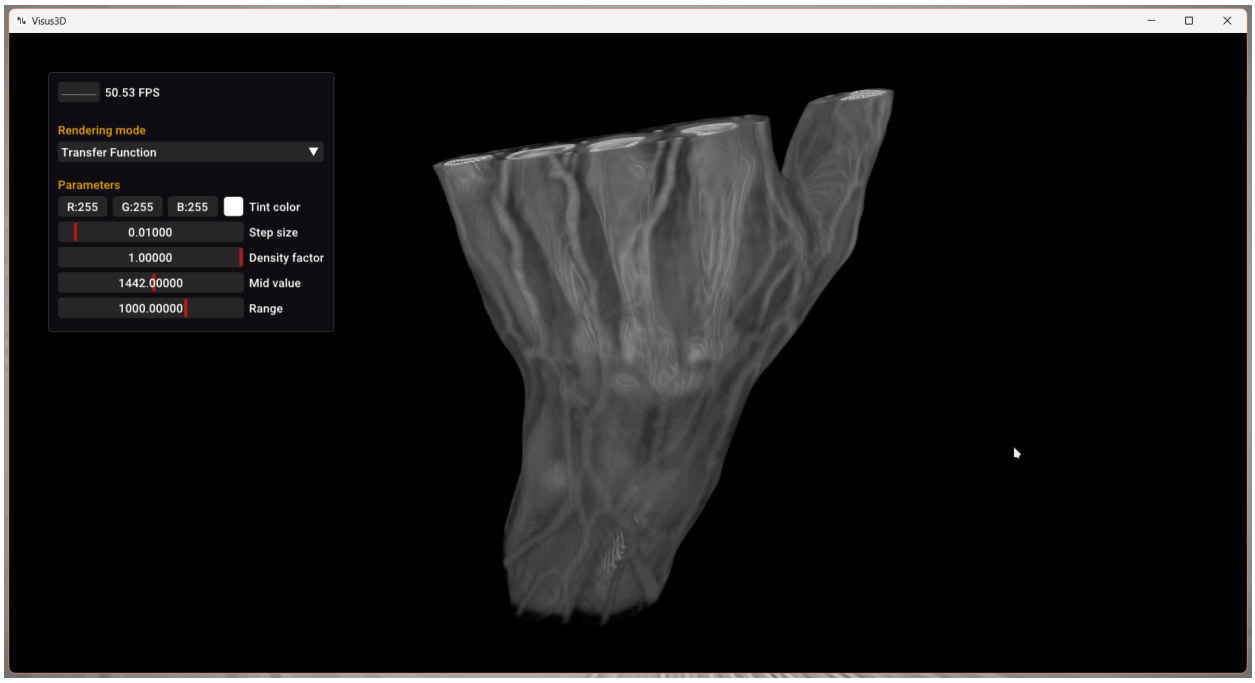


Figure 6.1: Visus3D: a GPU-based direct volume renderer.

Fig. 6.2 displays the different rendering modes.

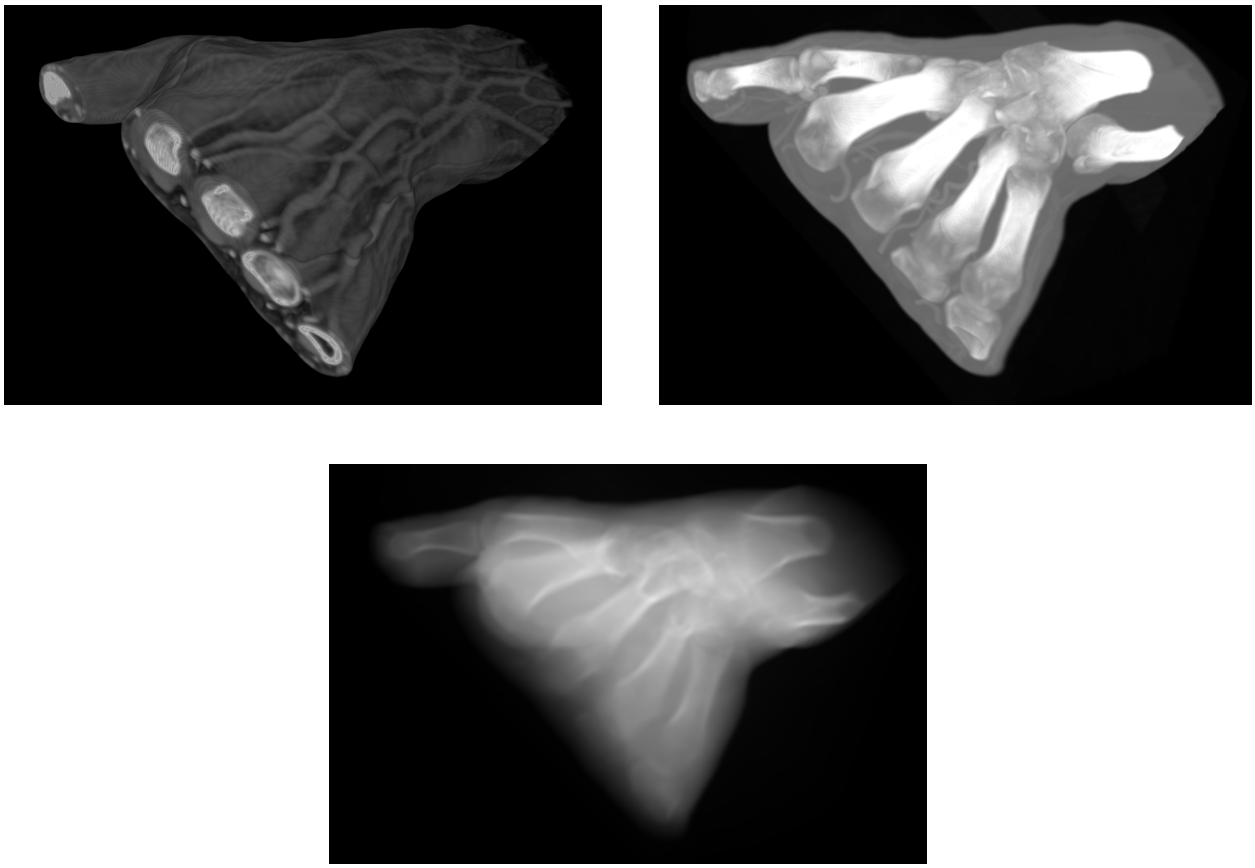


Figure 6.2: Transfer function (top-left), high density (top-right) and pseudo X-Ray (bottom).

Notice how changing the step size affects the sampling rate of the volume (see Fig. 6.3).

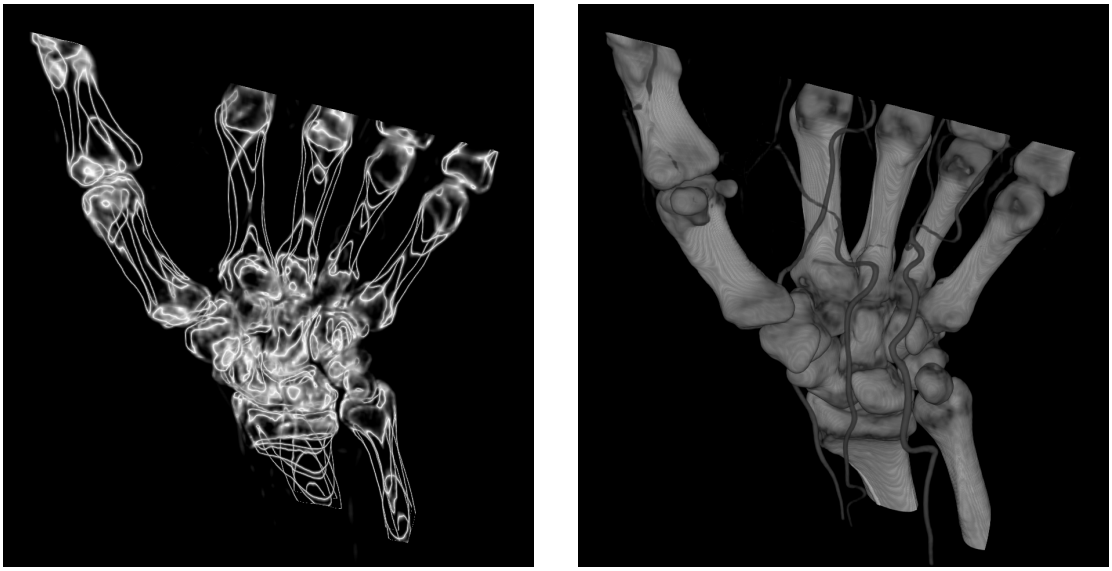


Figure 6.3: Low (left) vs high (right) sampling rate.

By playing around with the available parameters, we can isolate tissues and get very different

outputs, as is illustrated in Fig. 6.4.

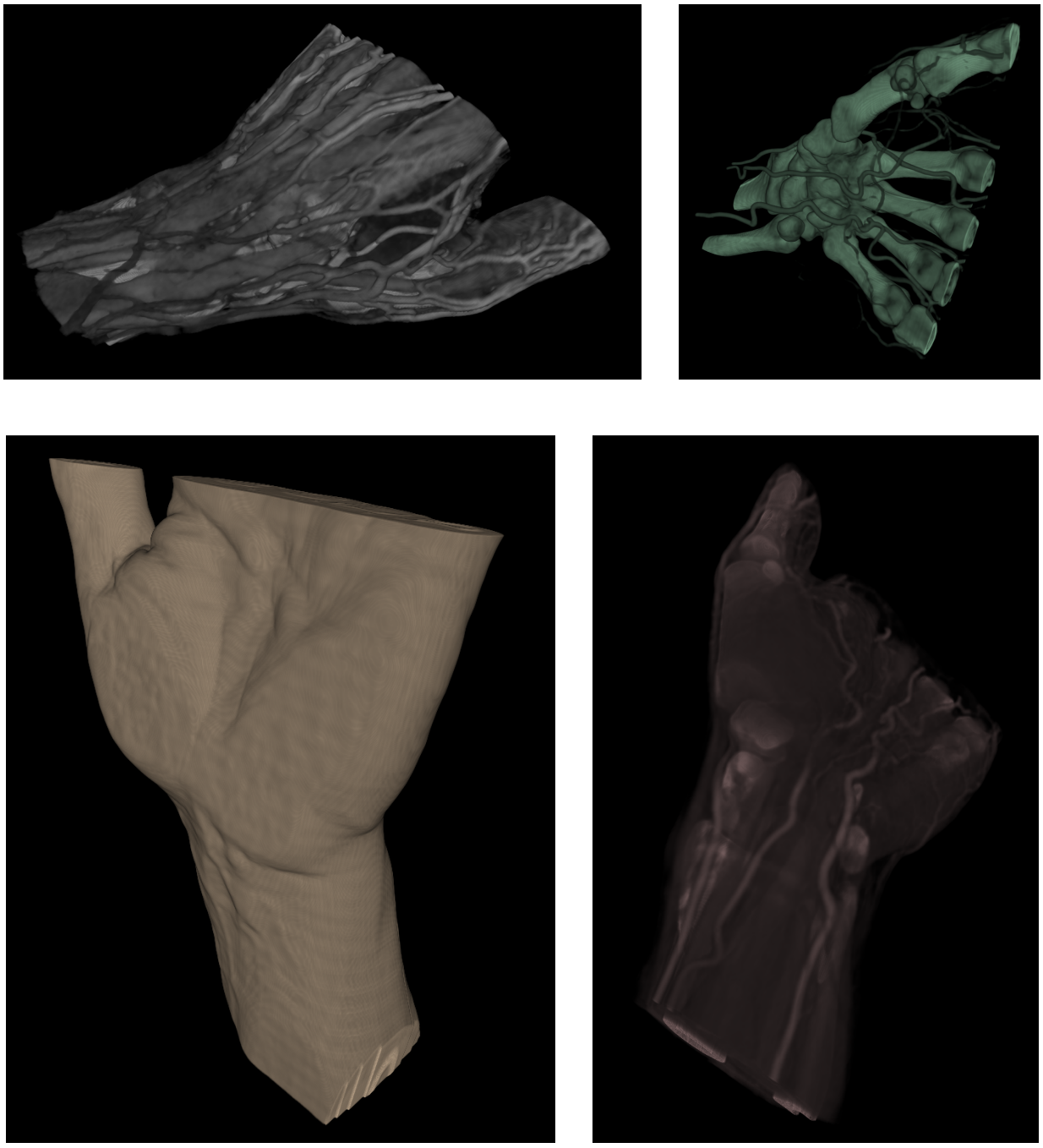


Figure 6.4: Various renders with different transfer function parameters.

6.2 Usability test

To evaluate usability, an interview was conducted with a sample of 8 people (including two nurses and a podiatrist). Prior to formulating the questions, we verbally explained the objective of the application and how to work the camera controller using the mouse (rotating, zooming and panning).

While we did mention that the floating GUI window could be clicked on and dragged around, we did not explain individual visualization parameters. The reasoning behind this was to provide just enough context to the users to be able to go on and explore on their own, so as to avoid influencing the interviewees' opinions and improve reliability in the results.

Below you can find the posed questions along with pie chart representations of the pool of answers (see Fig. 6.5 to 6.8).

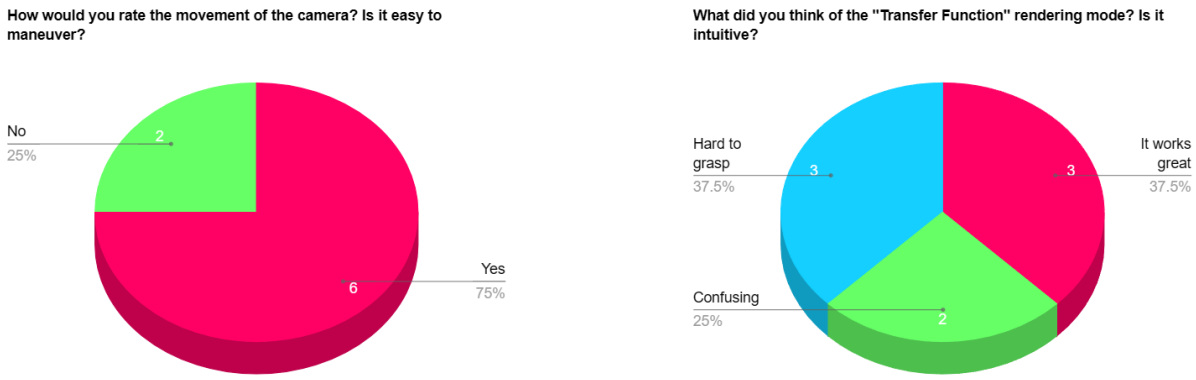


Figure 6.5: (Left): Camera. (Right): Transfer function rendering mode.

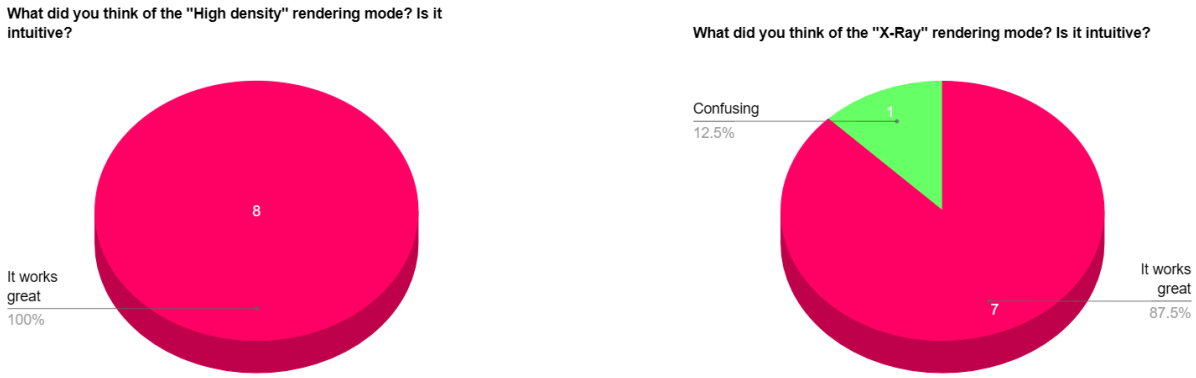


Figure 6.6: (Left): High density rendering mode. (Right): X-Ray rendering mode.

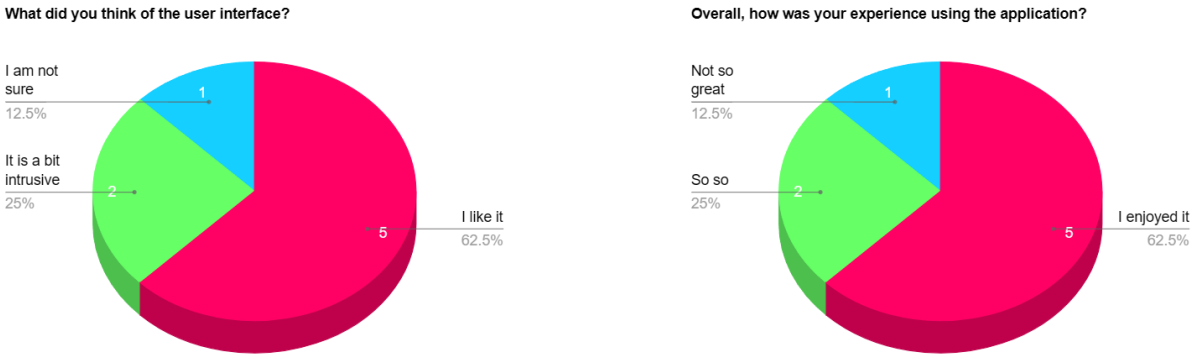


Figure 6.7: (Left): User interface. (Right): User experience.

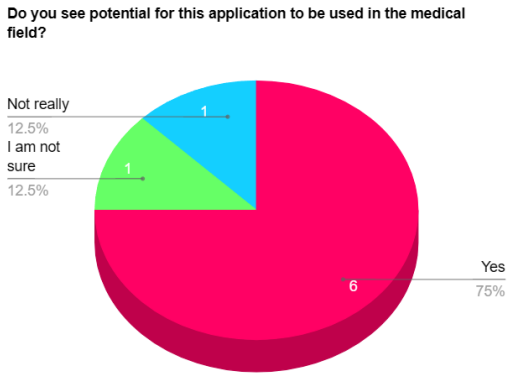


Figure 6.8: Foreseen potential.

After evaluation, we have concluded that the transfer function mode was specially difficult to grasp for non-specialist users. This is to be expected, given that the application is meant for experts of different scientifc domains. Also, one user suggested icons to switch between the different rendering modes, and another icon or a shortcut to toggle the visibility of the UI on and off. One of the nurses pointed out that he would like to have lighting and the ability to slice the volume, in order to better understand form and inspect cross-sections, respectively. The podiatrist also had some remarks, as he wanted multiple viewports with synchronized Multiplanar Reconstruction (MPR) support [23]. This is important to thoroughly inspect the volume in axis-aligned 2D slices in combination with the 3D render view.

Chapter 7

Conclusions

7.1 Summary

The primary goal that we set out to do was to develop a desktop application capable of visualizing volumetric datasets interactively, with a focus on medical imaging. In doing so, we defined a set of requirements which have, for the most part, been accomplished:

- **Scene navigation:** we implemented an arcball camera to orbit around the volume.
- **Data deserialization:** we used NIfTI binary files and C libraries to parse them.
- **Direct volume rendering:** three different modes of visualization were implemented (maximum intensity projection, average intensity projection and transfer function based compositing).
- **Optimizations:** we implemented early ray-termination, but could not perform empty space skipping, as the bounding volume is defined in the fragment shader itself. However, we used a single-pass raycasting algorithm, meaning we only have one draw call per frame.

Afterwards, we extensively tested the application, comparing render outputs with different settings and inspecting them visually. Here are some interesting findings:

1. Sampling rate plays a massive role in visual quality. Unfortunately, small step sizes severely hindrance performance, resulting in low frame rates.
2. A low sampling rate yields severe artifacts (wood-grain, also known as banding). This is again due to undersampling the volume. Sudden changes in depth happen between neighbouring opaque fragments belonging to the same surface.
3. Sparsely sampled volumes get brighter and densely sampled ones get darker. It would be worth looking into opacity correction when blending the fragments.

At the end, we conducted a usability test to find out the intuitiveness and usefulness of our work.

7.2 Future work

The original vision for the application has been realised, but it can be further improved by factoring in reviews/suggestions from our usability test, as well as adding new features. In no particular order, some of these are:

- Toggling the GUI on and off.
- Adding multiviewport support, to explore the volume from different angles and, possibly, using different rendering modes at once.
- Multiplanar reconstruction, to inspect voxel data per-slice [23].
- Stochastic jittering, to diminish wood-grain artifacts for not-so-low sampling rates. This technique adds small offsets (using a random factor) to the sampling locations of the viewing ray.
- The ability to change the background color, for enhanced contrast.
- Saving out renders to image files.
- As is often the case when working with images, an histogram representation can come in handy to analyze its distribution of values.
- Volume slicing, either with an axis-aligned / free plane or magic lenses (3D mesh filters). This can improve Focus+Context [1].
- Advanced transfer function strategies. The current one is not very flexible. We could add an indefinite number of control points that map different color and opacity tuples to various density values. 2D transfer functions are also a possibility.
- Proper bounding volume and spatial data structures such as octrees, so that we may perform empty space leaping.
- Advanced shading models with control over light placement, translucency, etc.
- Multi-volume support, to inspect different volumes side by side.
- Style transfer. We could apply predefined lookup tables for textures of known materials in the transfer function. This is a cost-effective way to emulate the different tissue surfaces.
- Post-processing effects: exposure, white balance, gamma correction, edge detection, etc.
- Indirect volume rendering and hybrid methods.

7.3 Connection to the academic curriculum

Physical Foundations: physical laws and models, energy conservation, oscillations and waves. It aided in understanding the wavelike behaviour of light and its optical properties, which are fundamental for realistic image synthesis.

Matrix Algebra and Geometry: it provides the mathematical backbone that all of computer graphics stands on.
Calculus. Differential Equations: differential and integral calculus, numerical integration, interpolation. Invaluable to grasp the Radiative Transfer Equation, the Volume Rendering Integral and interpolation schemes.
Introduction to Digital Signal Processing, Digital Signal Processing: sampling and filtering, Nyquist rate applies not only to temporal frequencies, but also to spatial frequencies (images).
Digital Image Processing. Computer Vision: instrumental knowledge about how images are stored, as well as image processing techniques. This was very helpful in understanding medical imaging formats and manipulating pixels in the fragment shader.
Programming 1, Programming 2: basic programming concepts, object oriented design, algorithmic thinking and version control systems.
Implementation of an electronic project using CDIO methodology: structured approach to planning, conceiving and executing the development of the project.
English for Software Developers: technical language, jargon, related to computer graphics and software development. Academic publications are predominantly written in English.

Bibliography

[1] InfoVis:Wiki, "Focus-plus-context." https://infovis-wiki.net/wiki/Focus-plus-Context, 2013. Accessed: 2024-07-09.

[2] G. T. Herman and H. K. Liu, "Three-dimensional display of human organs from computed tomograms," *Computer Graphics and Image Processing*, vol. 9, no. 1, pp. 1–21, 1979.

[3] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of ACM SIGGRAPH 1987*, pp. 163–169, 1987.

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proceedings of ACM SIGGRAPH 1988*, pp. 65–74, 1988.

[5] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, 1988.

[6] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware," tech. rep., University of North Carolina at Chapel Hill, 1993.

[7] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*, 1994.

[8] J. Kruger, R. Westermann, and T. Ertl, "Acceleration techniques for gpu-based volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, pp. 7–18, January–March 2003.

[9] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, July 2023.

[10] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," in *ECCV*, 2020.

[11] A. Yu, S. Fridovich-Keil, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance fields without neural networks," 2021.

[12] J. T. Kajiya, "Ray tracing volume densities," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 165–174, 1984.

[13] E. Angel and K. Moreland, "Fast high accuracy volume rendering," 2004.

[14] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and W. D., *Real-time Volume Graphics*. A K Peters, Ltd., 2006.

[15] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical evaluation of popular volume rendering algorithms," in *Proceedings of the 2000 IEEE Symposium on Volume Visualization, VVS '00*, pp. 81–90, 2000.

[16] A. Kaufman, "Voxels as a computational representation of geometry," in *The computational representation of geometry. SIGGRAPH 94*, 1994.

[17] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski, "Advanced illumination techniques for gpu volume raycasting," in *ACM SIGGRAPH ASIA 2008 courses, SIGGRAPH Asia '08*, (New York, NY, USA), pp. 1:1–1:166, ACM, 2008.

[18] J. Joshua Schpok, J. Simons, E. D., and H. C., "A real-time cloud modeling, rendering, and animation system," in *Eurographics/SIGGRAPH Symposium on Computer Animation*, Purdue Rendering and Perceptualization Lab, Purdue University and Scientific Computing and Imaging Institute, School of Computing, University of Utah, 2003.

[19] K. Shoemake, "Arcball: a user interface for specifying three-dimensional orientation using a mouse," 1992.

[20] G. P. Compendium, "Chapter 2: The graphics pipeline." https://graphicscompendium.com/intro/01-graphics-pipeline, n.d. Accessed: 2024-07-11.

[21] Wikipedia contributors, "Bit blit." https://en.wikipedia.org/wiki/Bit_blit, 2024. Accessed: 2024-07-15.

[22] J. Santell, "Model view projection." https://jsantell.com/model-view-projection/, 2019. Accessed: 2024-07-20.

[23] F. Fortin, L. McKay, R. Chieng, *et al.*, "Multiplanar reformation (mpr)," *Radiopaedia.org*, 2024. Accessed: 2024-07-28.

[24] J. Pawasauskas, "Ray casting." https://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm, 1997. Accessed: 2024-07-09.

Appendix A

User manual

This appendix serves as a cheatsheet of sorts, to assist you in effectively navigating and using the application. It provides detailed and clear instructions for all possible interactions with the software.

A.1 Launching the application

The application is delivered as an executable, along with DLL files and other necessary resources. To launch the application you simply need to run the executable by double clicking on it. Both left mouse button and right mouse button clicks work by default on Windows operating systems.

A.2 Camera manipulation

All camera patterns are mapped to the computer mouse input:

- To rotate the view, click and hold the left mouse button while you move the cursor around.
- To zoom the view, you can either scroll the mouse wheel up and down or click and hold down the wheel while you move the cursor around.
- To pan the view, you can click and hold the right mouse button while you move the cursor around.

A.3 Volume Rendering Settings

A.3.1 Rendering mode

There are three modes of visualization. By default, *Transfer Function* should be active. To change it, click on the drop-down menu with that name and select a different mode. You may repeat these steps to change modes back and forth.

A.3.2 Parameters

To change parameter values you may click and hold with the left mouse button on the red bar of the slider next to the parameter of your choosing and move it to the left or to the right, decreasing and increasing the value respectively. Note that this is also possible for the *R*, *G*, *B* color values. Additionally you may click over any of these with the right mouse button to change the color representation: *RGB*, *HSV* or *Hex*. Perhaps a more intuitive way to change the *Tint color* is by clicking on the colored squared next to its name with the left mouse button. A color selector should appear, allowing you to manually pick a color of your liking.

NOTE: bare in mind that if the floating GUI window gets on the way of the volume it can be moved around the application window space. To do so, click and hold with the left mouse button within the window, but outside any actionable parameter widget and drag the cursor to your desired location.

Appendix B

Sustainable Development Goals

Finally, we assess the project in the context of the Sustainable Development Goals (SDGs), highlighting its level of contribution, if any.

Adopted by all member states of the United Nations in 2015, the 2030 Agenda sets out a vision for sustainable development grounded in international human rights standards. Central to this framework are its accompanying SDGs, also known as the Global Goals, which provide a shared blueprint for peace and prosperity for all, now and into the future. These urgently call for action to tackle critical issues such as poverty, hunger and environmental protection.

ID	SDG	High	Moderate	Low	N/A
1	No Poverty.				X
2	Zero Hunger.				X
3	Good Health and Well-being.		X		
4	Quality Education.				X
5	Gender Equality.				X
6	Clean Water and Sanitation.				X
7	Affordable and Clean Energy.				X
8	Decent Work and Economic Growth.		X		
9	Industry, Innovation and Infrastructure.	X			
10	Reduced Inequality.				X
11	Sustainable Cities and Communities.				X
12	Responsible Consumption and Production.				X
13	Climate Action.				X
14	Life Below Water.				X
15	Life on Land.				X
16	Peace and Justice Strong Institutions.				X
17	Partnerships for the Goals.			X	

Table B.2: Assessment of the project's contribution to the SDGs.