PhD Thesis

Real-time Quality Visualization of Medical Models on Commodity and Mobile Devices

Jesús Díaz-García

Advisor: Pere-Pau Vázquez Alcocer
Co-advisor: Isabel Navazo Álvaro

Barcelona, April 2018

PhD Programme in Computing
Universitat Politècnica de Catalunya
Department of Computer Science (CS)

ViRVIG Research Group
Visualització, Realitat Virtual i Interacció Gràfica

Abstract

In the medical environment, imaging devices are able to capture data of biological tissue for their diagnosis. The visualization of such data in 3D is addressed by the so-called *Volume Rendering* or *Volume Visualization* techniques. However, these techniques are challenged by the continuous increase in the resolution of the datasets produced by modern capture devices.

Thanks to the evolution of graphics chips, most modern workstation/desktop *Graphics Processor Units* (GPUs) are able to handle these datasets without problem with classical visualization algorithms. Furthermore, thanks to their incredible market penetration and their improvement in performance and display capabilities in the last years, devices such as mobile phones or commodity hardware are becoming more and more usual candidates for diagnosis tasks in clinics. However, the capabilities of such devices are still far from their desktop counterparts, and applications running classical algorithms on these less powerful devices do not achieve the minimum desirable requirements regarding performance and quality. There is always a compromise between these two requisites. On the one hand, interactivity can be increased at the expense of decreasing the workload, sacrificing resolution and quality. On the other hand, enhancing the visualization quality involves performing a more intensive computation task that usually kills performance.

This thesis focuses on the rendering of volume datasets on nowadays' graphics devices with limited capabilities, such as those included in mobile devices, tablets or modest laptops. More precisely, we contribute with new approaches to allow the interactive generation of renderings and providing high-quality results while still not requiring long pre-processing times. To achieve these goals, we explore areas in the visualization pipeline such as the downsampling filter, the Transfer Function, and the visualization algorithm.

Multiresolution schemes are the most common way to address the problem of data size in environments with restrictive capabilities. Typically, volume data is reduced or simplified in order to provide real-time or interactive exploration during visualization. However, data reduction is accompanied by loss of features which can provide interesting details and information. In order to alleviate this data loss, we present a novel downsampling filter for volume datasets aimed at the preservation of fine details.

The quality of the images generated from downsampled models is greatly affected by the shading process, which mainly depends on the way in which gradients are computed. The well-known strategy of computing gradients on-the-fly from the downsampled model at the time of visualization generates visible artifacts and noticeable differences with respect to the rendering of the original dataset. To solve this, we approach this issue by using pre-computed gradients, and we propose a downsampling filter and an encoding scheme for this kind of data so that the gradients used for shading during the render of the low-resolution dataset better resemble the gradients computed from the original, high-resolution dataset.

Another aspect involving the quality of the images generated from downsampled datasets is the Transfer Function. Transfer Functions are designed to reveal specific information and interesting details of a particular dataset with its own histogram. However, the histogram of a downsampled dataset differs with respect to the original dataset. Based on this observation, we propose *Adaptive Transfer Functions*, an algorithm that, given the original Transfer Function and based on the differences between the original and the downsampled datasets, automatically generates fitted Transfer Functions that improve the quality of the renderings of these low-resolution representations.

Finally, and with the intention to put a stepping stone into what the future of medicine should be, *Mobile Health* (i.e., the creation of applications suitable for mobile and tablets), we focus on the visualization of volumetric models on mobile devices. In such environments, visualizations are expected to provide the highest quality and maximum resolution available. Moreover, in these applications, after exploring a particular model to search for a specific region of interest, it is interesting to obtain a detailed rendering of the selected view. We have addressed the problem of efficient generation of high-resolution images without compromising interactivity. In this field, we propose a multiresolution rendering scheme for volume models that uses a low-resolution model during user interaction and a high-resolution dataset for quality visualizations when the camera stops. Based on this architecture, we present then, two new progressive ray casting methods that allow the incremental rendering from low-resolution images to high-resolution images without compromising interactivity.

Resumen

En entornos médicos, los dispositivos de adquisición de imagen médica son capaces de capturar datos de tejido biológico para su diagnosis. La visualización de este tipo de datos en 3D se lleva a cabo con técnicas conocidas como *Volume Rendering* o *Volume Visualization*. Sin embargo, estas técnicas son desafiadas por el continuo aumento de la resolución de los datos producidos por los dispositivos de adquisición más modernos.

Gracias a la evolución del *hardware*, las estaciones de trabajo y las unidades de procesamiento de gráficos (GPUs) más modernas son capaces de manejar estos datos sin problemas con algoritmos clásicos de visualización de volumen. Por otra parte, en los últimos años, gracias a su increíble disponibilidad en el mercado y a sus mejoras en factores como su rendimiento y sus capacidades gráficas, los dispositivos móviles están volviéndose, cada vez más, en candidatos para las tareas de diagnosis en muchas clínicas. Sin embargo, las capacidades de estos dispositivos todavía están lejos de los equipos de sobremesa. En ese sentido, muchas aplicaciones que utilizan algoritmos clásicos de visualización, no funcionan satisfactoriamente en este tipo de dispositivos menos potentes, en lo que se refiere a calidad y rendimiento. Siempre hay un compromiso entre estos dos requisitos. Por una parte, el grado de interactividad se puede aumentar si se sacrifica la calidad de los resultados. Por otra parte, mejorar la calidad gráfica requiere ejecutar cálculos más intensivos que normalmente reducen el rendimiento.

Esta tesis se centra en la visualización de modelos de volumen en los dispositivos gráficos de hoy día que son menos potentes. Ejemplos de estos dispositivos son los teléfonos móviles, tablets y portátiles de baja gama. En particular, nuestras contribuciones en esta tesis son métodos que permiten la visualización interactiva de modelos de volumen y que proporcionan resultados de buena calidad, sin necesitar largos tiempos de cálculo o pre-proceso. Para conseguir estos objetivos, se exploran areas en la *pipeline* de visualización tales como el filtrado de *downsampling*, la Función de Transferencia, y el algoritmo de visualización.

Las técnicas multiresolución son el modo más común de enfrentarse al problema de los datos grandes en entornos con capacidades limitadas. Normalmente, los datos de volumen son reducidos o simplificados para conseguir una exploración en tiempo real o interactiva durante la visualización. Sin embargo, la reducción de datos viene de la mano de una pérdida de información, con lo cual, características y detalles interesantes se pueden perder en este proceso.

Para minimizar esta pérdida de datos, presentamos un novedoso filtro de *downsampling* para datos de volumen diseñado para preservar detalles del modelo original.

La calidad de las imágenes generadas de los modelos simplificados tambin se ve afectada en gran medida por el sombreado. La operación de sombreado depende principalmente del modo en el que los gradientes del campo escalar se han calculado. La conocida estrategia de calcular los gradientes *on the fly*, directamente a partir del modelo de baja resolución en el algoritmo de visualización, genera artefactos visibles y diferencias notables con respecto a visualizaciones del modelo original. Para solucionar esto, proponemos una solución basada en gradientes precalculados, y proponemos un filtro de *downsampling* para gradientes y un esquema de codificación para maximizar el número de direcciones de gradientes representables, de modo que los gradientes obtenidos para el modelo de baja resolución se parezcan más a los gradientes del volumen de alta resolución.

Otro aspecto importante en cuanto a la calidad de la visualización de modelos reducidos es la Función de Transferencia. Las Funciones de Transferencia son diseñadas para revelar información específica y detalles interesantes de un modelo de volumen en particular. Sin embargo, el histograma de un modelo en particular difiere con respecto al histograma de cualquiera de sus modelos simplificados. A partir de esta observación, proponemos *Adaptive Transfer Functions*, un algoritmo que, dado un modelo de volumen y una Función de Transferencia diseñada para éste, genera automáticamente Funciones de Transferencia específicas para los modelos simplificados, de modo que la calidad de las visualizaciones de baja resolución mejoran notablemente.

Por último, con la intención de entrar en el futuro de la visualización en el ámbito de la medicina (i.e., la creación de aplicaciones adecuadas a móviles y tablets), nos centramos en la visualización de modelos de volumen en teléfonos móviles. En entornos médicos, es necesario generar imágenes de alta calidad con la máxima resolución posible. Por ese motivo, en estas aplicaciones, después de explorar un modelo en particular para buscar una región de interés específica, es interesante obtener una imagen lo más detallada posible de la vista seleccionada. Nosotros hemos tratado este problema a través de un esquema multiresolución de visualización que utiliza un modelo de baja resolución mientras el usuario de la aplicación está interactuando, y un modelo de alta resolución para proporcionar visualizaciones de calidad cuando la cámara para. Basándonos en esta arquitectura, presentamos dos nuevos métodos de pintado de volumen que permiten la generación incremental de imágenes pasando de la baja a la alta resolución sin comprometer la interactividad.

Agradecimientos

Había momentos en los que parecía que no se iba a terminar, pero ahora parece que sí, que... ¡ya se acaba! Es curioso como cambian las cosas con el tiempo y según la perspectiva. ¿Y cómo es un doctorado? Para la gente que como yo "es de gráficos", seguro que entenderán si digo que la respuesta es *view-dependent*. La ilusión, los ánimos, y la energía fluctúan. Muchas cosas van cambiando. Uno mismo cambia. La función tiene máximos y mínimos, pero lo importante es que si la paso por un filtro *Gaussiano* (mejor bilateral, no vayamos a perder esos contrastes) el resultado es muy positivo. Esta página es para quienes han hecho subir la media.

Primero y por encima de todo, tengo que dar infinitas gracias a mis tutores, Isabel Navazo y Pere-Pau Vázquez, por haberme ayudado tanto en todo momento con sus conocimientos, sus ánimos, su paciencia conmigo y su guía. De igual modo, transmito mi gratitud a Pere Brunet, que aunque no oficialmente, es mi tercer tutor, y como tal ha contribuido en la misma medida con su tiempo y dedicación. Gracias a los tres por vuestra labor como tutores y por ser bellísimas personas.

Gracias a la UPC y a mi grupo de investigación, ViRVIG, por el soporte y los medios que me han proporcionado para poder llevar a cabo la tesis. Gracias al resto de profesores que en algún momento me han ayudado con el trabajo, en formulaciones matemáticas, en correciones de inglés, con opiniones para mejoras y con halagos por el trabajo hecho. Gracias a mis amigos del CRV, a los que se ya se fueron y a los que aún siguen, por haber compartido tantos momentos juntos, en el puesto de trabajo y fuera de él, mientras programábamos o mientras jugábamos *online*, hablando de trabajo o hablando de la vida, en definitiva, por hacer del despacho un lugar tan familiar.

A Alma IT Systems por los primeros años de apoyo en mi tesis. En especial, quiero mostrar mi agradecimiento a Frederic Pérez por su apoyo y ayuda en esa etapa. Por todas las reuniones que hicimos para revisar el trabajo hecho y encontrar nuevas direcciones e ideas para mejorar nuestro trabajo.

Y por descontado, gracias a los míos. A mi hermana, Marina, por la portada tan chula que le ha puesto a esta tesis. A mi familia y a mis amigos, a los más cercanos, que me han arrancado una sonrisa en un día de bajón, o me han soportado cuando me rechazaban un artículo, o ¡celebraban conmigo cuando me lo aceptaban!, o simplemente están o han estado ahí conmigo, pasando el rato, mirando la tele, comiendo guarradas, yendo a bailar, yendo a hacer deporte... cuidando de mi.

Contents

1 Introduction

Since the beginning of time, humans have felt the creative impulse of reflecting their experiences and observations through the use of images. This has been observed in history since the Paleolithic, where the most ancient cave paintings dated to some 40.000 years ago. With the passage of time, there have been more and more shreds of evidence, as seen in the Egyptian hieroglyphs, the Greek geometry, and Leonardo da Vinci's revolutionary methods of technical drawing for engineering and scientific purposes.

With the arrival of modern times, the evolution of technology brought with itself computational devices that were able to perform engineering and mathematical tasks that were not easily solved before (or not possible at all). These devices evolved up to the point of becoming nowadays' modern computers and made possible the ever-growing development of *Computer Science*, which studies the automation of algorithmic processes of different nature.

Approaching the discipline concerning this thesis, *Visualization* is a subfield of *Computer Science* which studies methods and techniques for the generation of images in order to effectively communicate a message or to provide some kind of information. Within this field, *Scientific Visualization* is a branch of science primarily concerning the realistic graphical representation of three-dimensional phenomena. Used in many situations with the aim of *Scientific Visualization*, *Volume Rendering* [34] is a set of techniques that generate images by means of projecting datasets that capture this information of volumetric nature onto a 2D image.

In the last years, the appearance of *Graphics Processing Units* (GPUs) has prompted the implementation of *Direct Volume Rendering* (DVR) algorithms, which consist in generating an image directly from a volume dataset, allowing for real-time or interactive visualization applications. For scalar fields, this image generation is typically achieved by mapping the scalar values within the volume domain to color data by means of a *Transfer Function* (TF) which is typically represented by a piecewise linear function describing a color table, although it may take more complex forms. In the case of DVR, both the volumetric scalar field or dataset and the TF are usually stored in the internal memory of the GPU as a 3D texture and a 2D texture, respectively. Furthermore, the way the color is composed to make the projected 2D image depends on the rendering technique, which can also be primarily implemented within the graphics processor using shader programs. Some of the most known DVR techniques are splatting, shear-warp, slicing, and ray casting, being the latter the state of the art algorithm for *Volume Rendering*, due to its parallel nature that allows to efficiently translate its implementation to GPUs and also due to the quality of the images it generates.

Volume Rendering techniques can be very convenient in many fields of research. Some examples are the comprehension of the airflow around planes and cars in aerodynamics, the analysis of seismic data for the study of terrain in geoscience, the understanding of meteorological data or the medical imaging for the clinical practice. *Medical Visualization* in particular, is a sub-field of *Volume Rendering* aimed at the display of 3D medical images to ease specialists' daily tasks such as clinical diagnosis, surgery planning, and medical education.

1.1 Motivation

Mechanisms for medical imaging acquisition such as *Magnetic Resonance Imaging* (MRI), *Computerized Tomography* (CT) and micro-CT scanners are continuously evolving, up to the point of obtaining volume datasets of large resolutions ($\geq 512^3$). These datasets are typically composed of a stacked set of 2D slices. As these datasets grow in resolution, its treatment and visualization become more and more expensive due to their computational requirements. Thus, special techniques such as data pre-processing (filtering, construction of multiresolution structures, etc.) and sophisticated algorithms have to be introduced in different points of the visualization pipeline in order to achieve the best visual quality without compromising performance times.

Managing big datasets becomes a problem in scenarios with limited computational resources. In the last years, *Volume Visualization* has only been possible for some medium-large datasets in real time, thanks to the computational power of newer GPUs. However, dealing with big amounts of data without taking into account special considerations, would require having enough memory storage and processing power available. The bigger the dataset, the more demanding these requirements are. For that reason, standard methods for *Volume Visualization* that have worked well with medium-large datasets, are not suitable for those huge datasets on all hardware.

Not long ago, the only physicians that were using 3D medical visualization tools were radiologists. Nowadays, the outcome of diagnosis is the data itself, and medical doctors need to inspect them in commodity PCs (even patients may want to render the data, and the DVDs are commonly accompanied with a DICOM viewer software, which empowers patients in their health management). Furthermore, with the increasing use of technology in daily clinical tasks, small devices such as mobile phones and tablets can fit the needs of medical doctors in some specific areas. Visualizing 3D diagnosis images of patients becomes more challenging when it comes to using these devices instead of desktop computers, as they generally have more restrictive hardware specifications.

1.2 Thesis statement

The goal of this thesis is the quality, real-time visualization of medium to large medical volume datasets (resolutions $\geq 512^3$ voxels) on mobile phones and commodity devices.

To address this problem, we use multiresolution techniques that apply downsampling techniques on the full resolution datasets to produce coarser representations which are easier to handle. We have focused our efforts on the application of *Volume Visualization* in the clinical practice, so we have a special interest in creating solutions that require short pre-processing times that quickly provide the specialists with the data outcome, maximize the preservation of features and the visual quality of the final images, achieve high frame rates that allow interactive visualizations, and make efficient use of the computational resources.

1.3 Addressed problems and contributions

The contributions achieved in this thesis comprise improvements in several areas of the rendering pipeline. The techniques we propose are meant to improve the stages of multiresolution generation, Transfer Function design and the GPU ray casting algorithm itself. The following paragraphs briefly describe the general idea of these contributions:

- **Improved feature-preserving downsampling filtering for scalar fields.** In Chapter 4 we present an evaluation of different downsampling filters used to generate coarser representations of the original dataset, and we analyze their effectiveness at preserving details. Moreover, we propose a new Gaussian-based, feature-preserving filter that produces quality low-resolution representations and conserves small features that are prone to disappear during the downsampling process [15, 17].

- **Improved downsampling and efficient storage of gradient data.** One of the factors that most contribute to the final image quality in volume rendering is proper shading. Downsampling also affects the way in which gradients are computed from the scalar field. The most used method to compute gradients in standard GPU ray casting is performing the calculation on-the-fly at each sample step. However, this may lead to severe artifacts if the model has been downsampled aggressively. In Chapter 5, the effect of shading coarser datasets using different methods for the computation of gradients is explored. Furthermore, a consistent way to downsample and efficiently store pre-computed gradients is proposed [18].

- **Improved visualization of coarse datasets using Adaptive Transfer Functions.** When creating coarse models of a multiresolution hierarchy, the data loss resulting from the downsampling process affects the visualization quality. In particular, the Transfer Function originally designed for the original scalar field might be not valid anymore for coarser representations. In Chapter 6, we present Adaptive Transfer Functions [16], an algorithm that, by modifying the original Transfer Function, generates custom Transfer Functions for downsampled models so that the quality of renderings is highly improved. The technique is simple and lightweight, and it is suitable not only to visualize huge models that would not fit in a GPU, but also to render not-so-large models in mobile GPUs, which are less capable than their desktop counterparts.

Moreover, it can also be used to accelerate rendering frame rates by using lower levels of the multiresolution hierarchy while still maintaining high-quality results in a focus and context approach. We also show an evaluation of these results based on perceptual metrics.

- **Interactive high resolution rendering on mobile devices.** Mobile devices have experimented an incredible market penetration the last decade, and currently, medium to premium smartphones are relatively affordable devices. With the increase in screen size and resolution, together with the improvements in the performance of mobile CPUs and GPUs, more tasks have become possible, but DVR remains a challenge in such devices. We have explored the limitations of rendering from medium to large volumetric models in mobile devices. Not surprisingly, we have observed that the interactivity achieved by these devices is easily affected at the time of visualizing datasets of increasing resolution, eventually leading to application stalls and crashes even when the visualized model fits the memory capabilities of the device. To solve these problems, Chapter 7 presents two progressive ray casting methods that can obtain high-quality results on mobile devices for models that some years ago were only supported by desktop computers without compromising interactivity at all [19].

1.4 About this document

The remainder of this document is organized as follows: Chapter 2 introduces key concepts about *Direct Volume Rendering*, explains the core techniques used all along the development of this thesis and provides the foundations on which our contributions are based. This chapter also provides an overall view of the visualization pipeline and briefly reviews its stages, and identifies their shortcomings as a motivation to glue together the later chapters with the provided contributions. Chapter 3 gives an overview of the state of the art publications which are most related to the areas covered by the contributions of this thesis, mainly those explained in the previous section. The following four chapters explore in detail the studied subjects, the methods, and the results accomplished in the course of this thesis. First, Chapter 4 presents an analysis of several downsampling filters and proposes a novel feature-preserving downsampling filter for volumetric scalar fields. Chapter 5 proposes a downsampling filter and an encoding strategy for pre-computed gradient data that avoids common visualization artifacts when shading coarse datasets. In Chapter 6, we present an algorithm to improve the visualization of multiresolution datasets by means of generating an adapted version of the original Transfer Function for each coarse level of resolution. Chapter 7 addresses the problem of high-quality interactive rendering on mobile phones and commodity hardware, combines the previous contributions into a multiresolution framework with the aim of incremental rendering, and presents two algorithms for the progressive ray casting of volume datasets. Finally, the conclusions of this thesis are presented in Chapter 8.

2 Preliminaries

Before getting into any material related to the specific contributions of this thesis, this chapter will introduce the visualization pipeline and explain its different stages when applied to the visualization of medical models in restrictive hardware. It will review the acquisition mechanisms and the data structure used to store the captured information. Then, it will introduce the processing stage, where the data filtering and the multiresolution generation processes take place. Finally, going into the visualization stage, it will present the Transfer Function, and the specific rendering techniques used in the context of the ray casting algorithm. This quick overview of the pipeline will serve us as a starting point in order to identify the common flaws we will build on to provide the contributions given in the next chapters.

2.1 Visualization pipeline for 3D medical models

A visualization pipeline is a model that describes the visualization process of some kind of data, which involves several stages, each modeled by a specific data transformation. Figure 2.1 shows a schematic overview of a standard visualization pipeline used for volumetric models obtained from 3D image data. The stages depicted in the figure include data *acquisition*, data *processing* (filtering and multiresolution generation), and data *visualization* (mapping, and rendering).

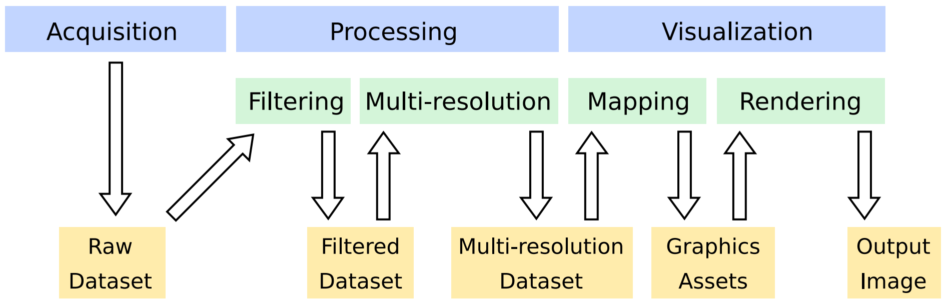


Figure 2.1: Schematic overview of a visualization pipeline for volumetric medical models. After its acquisition, raw data is processed, first filtered and then downsampled to generate a multiresolution hierarchy. Then, multiresolution datasets are mapped to GPU assets (textures, primitives, etc.), building a visual representation of the initial data that can be finally used by a rendering algorithm, to generate output images.

Acquisition

At the beginning of the process, data must be acquired from the biological tissue into a digital representation. The medical imaging acquisition devices are the responsible for capturing the information of specific body regions. Typical examples of this kind of techniques are *Computerized Tomography* (CT) or *Medical Resonance Imaging* (MRI). The captured dataset consists of a series of stacked images (slices of the captured anatomical part) that represent the volume, and its most used representations are regular 3D grids of samples (see Section 2.2). Each sample in these grids represents a single volume element called *voxel*, and typically contains the information related to the density of the material in that region.

Processing

Raw data obtained from medical acquisition devices are not usually in the most proper form for visualization. As a pre-process, data needs to be typically prepared (*filtered*) previously in order to remove noise and other imperfections coming from the acquisition process. Furthermore, depending on the target platform, data has to be also simplified (*multiresolution*) in order to fit the capabilities of the destination hardware to ensure interactive visualizations. This downsampling process results in the loss of some information. See Section 2.3 for details about multiresolution datasets.

Visualization

Three-dimensional datasets representing a scalar field cannot be directly visualized with the tools provided by nowadays' conventional graphics hardware. In order to transmit this information into the graphics pipeline, we must map the acquired datasets into a set of graphical primitives (*mapping*).

There are many strategies which can be followed in order to obtain a visual representation of a volume dataset (*rendering*). These strategies can be mainly classified in two blocks depending on the nature of the algorithm: indirect and direct volume rendering methods.

Indirect volume rendering extracts an intermediate representation that consists of a set of polygonal primitives (e.g., triangles) that will be rendered later using traditional polygon-based techniques. The extracted polygonal representation usually matches a surface conformed by all the points in the scalar field with a particular value (*isovalue*). Some of the most famous algorithms for indirect volume rendering are *Marching Cubes* [51] and *Projected Tetrahedra* [76]. The advantage of these methods is that the rendering step is usually very efficient. However, they do not allow interactive modification of the visualization criteria (the *isovalue*) to focus on different features of the source dataset.

Direct volume rendering (DVR) refers to those algorithms that directly resample the input dataset at many locations and project their visual representation to screen. This visual representation is obtained by simulating the interaction of the optical properties of the sampled material with light. With direct volume rendering, the data is considered to represent a semi-transparent light-emitting medium, so phenomena such as gaseous materials can be simulated, and the volume dataset is used as a whole, making it possible to show all interior structures. There are several methods that have been used for the purpose of DVR [34], such as *shear-warp*, *splatting*, *cell projection*, *texture slicing* and *ray casting*. Among these methods, the tendency is to move toward GPU implementations of the ray casting algorithm, which is the state of the art technique for DVR. For the rest of the thesis, we will work mainly on this visualization algorithm. A brief introduction to ray casting can be found in Section 2.6.

2.2 Acquisition process: voxel model

Typically, volume rendering of medical models assumes a continuous 3D scalar field function V that translates from 3D locations to scalar values. This function can be written as a mapping:

$$V(x) : \mathbb{R}^3 \rightarrow \mathbb{R}$$

However, in order to represent this three-dimensional scalar field into the discretized world of computers, it needs to be stored in a discretized grid, because it is the result of a measurement performed at a finite number of sample locations. Some typical grid structures used to represent discretized data are tetrahedral grids, distorted hexahedral grids, mixed prism grids, or uniform n-dimensional grids. The last representation, n-dimensional grids, (and more precisely 3D grids) are the most commonly used for volume rendering.

Medical imaging acquisition devices such as Computerized Tomography (CT), or Magnetic Resonance Imaging (MRI), typically provide a stack of 2D slices uniformly separated along a given axis, composing a uniform 3D grid that represents the scanned volume region. Each pixel of these 2D images (or sample in the 3D grid) contains a value representing a scalar property (density in CTs) in the surrounding volumetric region (see Figure 2.2).

Uniform grids contain samples that are typically evenly separated in the 3D space along each X, Y and Z axis. This spacing is potentially different in the

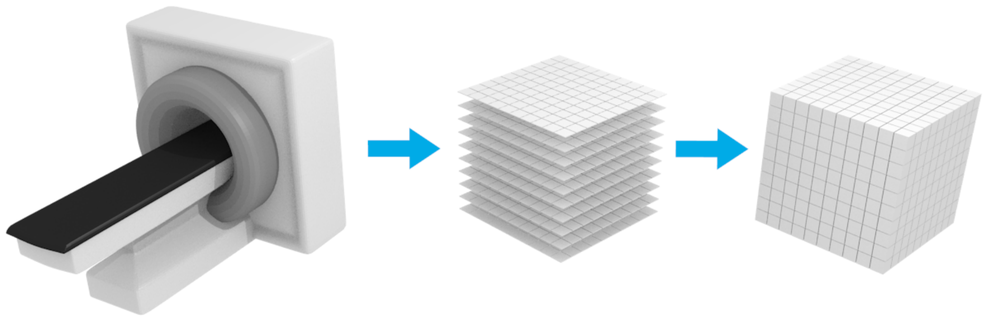


Figure 2.2: Medical imaging devices obtain a set of 2D slices containing properties of the anatomical tissue (e.g. its density, in the case of CTs). These slices, if stacked compose a 3D voxel model that represents the captured volume.

X, Y and Z axes, and is given by the resolution of the scanning or acquisition device used to make the measurement of the source data. Each one of the samples in the grid is called a *voxel*, or volume element, and the whole set of voxels (i.e. the stacked set of 2D images) is called *voxel model*. Along the rest of this document, other names that will refer to this representation may be *volume model*, *volume dataset* or simply *model* or *dataset*, indifferently.

An issue of concern about voxel models, which are essentially *discrete*, is that they are way far from representing a continuous scalar field V. For that reason, along with the interpretation of *voxel*, there must be a reconstruction filter: an interpolation scheme that fills the in-between space, allowing to evaluate values at arbitrary positions in the spatial domain $\mathbb{R}^3$. Unless otherwise stated, we assume the use of a tri-linear interpolation scheme to query density values from discretized grids. It provides an acceptably smooth reconstruction of the original scalar field, and it is usually an accelerated feature implemented in most current graphics hardware.

Another important consideration is that voxel models make it difficult to have available a complete statistical description of the underlying physical phenomena. The effects of *quantization* on the statistics are difficult to account for. Some studies have explored this field in order to obtain a better understanding of discrete representations. For instance, in [21], the authors develop a mathematical model of quantized statistics of continuous functions and prove convergence of geometric approximations to continuous statistics for regular rectilinear grids.

Despite the limitations mentioned just above (*discretization* and *quantization*), being well structured, and making possible a compact representation as 3D arrays in computer memory and fast access to random data cells, voxel models are the most common data structure in practical applications such as *Medical Visualization*. We will be assuming the usage of this data structure to store any volume dataset in the rest of the document.

2.3 Multiresolution and filtering

Because modern data acquisition methods have been able to produce datasets of big resolutions, volume rendering of these kinds of datasets has become a challenging task for some devices such as mobile devices or the commodity hardware available in clinics and hospitals. In medical visualization, typical

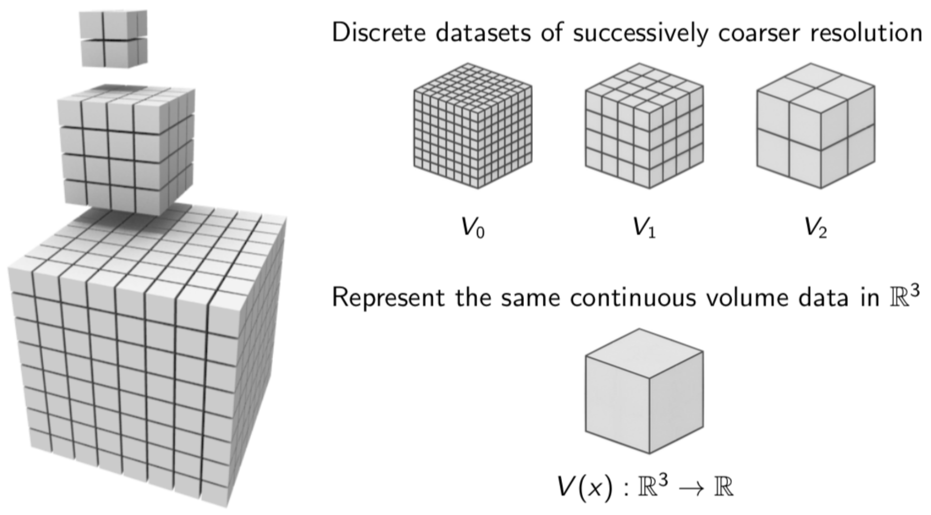

Discrete datasets of successively coarser resolution
V_0
V_1
V_2
Represent the same continuous volume data in $\mathbb{R}^3$
$V(x) : \mathbb{R}^3 \to \mathbb{R}$

Figure 2.3: A multiresolution hierarchy of volume dataset is a set of voxel models of successively coarser resolutions that represent the same scalar field V. The original dataset V_0 typically contains the raw data obtained by some capture devices after its acquisition. The lower resolution datasets $V_1 \dots V_n$ are obtained using downsampling methods.

studies provided by CT may easily consist of 2000 slices, with slices of 512×512 pixels, with a precision of 12bit each one. Voxel models of these dimensions ($512^2 \times 2000$) result in roughly a gigabyte of data. Other techniques such as MicroCT can even achieve higher resolution datasets, resulting in several gigabytes of data.

Several problems arise from big datasets. Mainly, due to the amount of data they contain, they have demanding memory requirements. These issues are not especially important regarding the storage capabilities of most devices. However, large datasets may not fit into the main GPU memory, which is scarce in mobile devices. Furthermore, transmitting big amounts of information require having large bandwidths in order not to increase the latency of data transport when this needs to be sent across a network link. Similarly, the same issue applies locally, at the time of uploading information from the storage device to main memory. In addition, in the hypothetical case that the memory size requirements are not a problem, having large amounts of information implies large amounts of work for the visualization algorithm, which means that the GPU will take longer to finish rendering a frame, decreasing the frame rate and consequently reducing the level of interaction.

In order to address these issues, one of the most known approaches is multiresolution. A multiresolution volume dataset is a set of datasets $V_0, V_1 \dots V_n$ of successively coarser resolutions (see Figure 2.3). The reduction factor could vary and could be applied differently to the different axes of the model. However, we will assume a reduction factor of two in each dimension from successive resolutions so that, for $k > 0$, V_k is stored in a 1/8th of memory required for V_{k-1}.

To generate a coarse dataset V_k in the hierarchy, the original (or a higher-resolution) dataset is usually filtered with a symmetric weighting function w of a finite domain. After filtering, the high-resolution model is then usually subsampled or resampled at a lower resolution to finally obtain the coarser resolution voxel model V_k.

A typical example of the usage of multiresolution is selecting a certain coarse dataset to be used for visualization whenever any other finer (or higher resolution) dataset in the hierarchy is too big to be properly managed by the target hardware (e.g., in terms of memory size or processing speed).

Inevitably, coarse representations in the multiresolution hierarchy lose information as the level of simplification increases. This point is one of the most critical stages in the visualization pipeline where information can be lost. Several contributions of this thesis focus on this issue from different perspectives. Chapter 4 in particular studies several filters for the generation of the multiresolution hierarchy (downsampling) and proposes a novel feature-preserving filter to achieve higher quality downsampled models.

2.4 Direct volume rendering

The target of *Direct Volume Rendering* (DVR) is to directly extract the visual information from a volumetric model V in order to reflect the physical properties of the participating medium it represents. To provide a volumetric description of its physical properties, we use an optical model that simplifies the computationally expensive task of solving the equations of light transport, which rely on complex geometric optics. The most used optical model, among others in DVR, is the *emission-absorption* model [36, 52], in which each particle of the volume can emit light and absorb incident light, but scattering and indirect illumination are neglected. It provides a good compromise between generality and efficiency of computation. In order to apply the *emission-absorption*

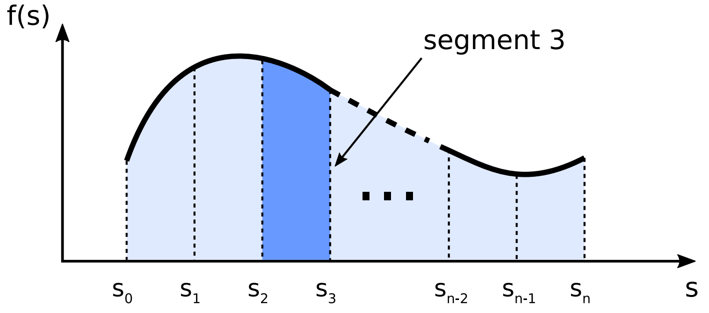


Figure 2.4: Partitioning of the integration domain into several intervals. The intervals are described by locations $s_0 < s_1 < \ldots < s_n$. The ith interval is the one between the locations $[s_{i-1}, s_i]$. This image is inspired by Figure 1.4 in [34].

model, two terms q and k that define the optical material properties are needed. These terms are the emission of light and the true absorption of the material at a certain point, respectively, as assigned through a Transfer Function (see Section 2.5. The *emission-absorption* model, when integrating along the direction of light towards the view position, leads to the *volume rendering integral* [34]:

$$I(D) = I_0 e^{-\int_{s_0}^{D} k(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} k(t)dt} ds. \tag{2.1}$$

where $I(D)$ is the intensity of light (*radiance*) leaving the volume at the position $s = D$ and finally reaching the camera. In the first term, I_0 represents the light entering the volume from the background at the position $s = s_0$, and the following exponential function is the attenuation it suffers as traveling through the participating medium of the volume from s_0 to D. The second term represents the integral contribution of the illuminated volume attenuated by the participating medium along the remaining distances to the camera.

The volume rendering integral cannot be computed analytically for most datasets. The most common approach to implement an approximation as close as possible to its solution is by splitting it into several integration intervals. These intervals can be described by locations $s_0 < s_1 < \ldots < s_n$, being s_0 the starting point of the integration, located at the back part of the volume, and $s_n = D$ is the endpoint, closer to the camera. Figure 2.4 illustrates the partitioning of the integration domain into several segments. With this partitioning, the volume rendering integral can be computed in a more friendly way for numerical methods as follows:

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j \qquad \text{being } c_0 = I_0. \tag{2.2}$$

where c_i is the color contribution of the segment going from s_{i-1} to s_i, and the initial case c_0 is the color contribution of the light I_0 at the point of entering the volume from behind, in the direction of the eye. T_j is the accumulated transparency from s_j to s_{j+1}. Note that although the previous formula is dealing with transparencies T_j, it is much more common in computer graphics in general (and so will be in the rest of the document) speaking about opacities $\alpha_j = 1 - T_j$.

The discretized volume rendering integral (equation 2.2) can be translated to a more suitable form for processor units. Iterative computation allows solving the equation by compositing. The idea is to split the summations and multiplications in equation 2.2 into several, yet simpler operations that are executed sequentially. The integral can be solved by iterating in two directions, which leaves two different composition schemes: front-to-back and back-to-front compositing.

When the viewing rays are traversed from the viewing point into the volume, the front-to-back scheme is applied:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned} \tag{2.3}$$

In this case, the resulting color and opacity after each compositing step (C_{dst}, α_{dst}) are calculated by adding the colors and opacities of the current interval (C_{src}, α_{src}) attenuated by the opacity of the front part of the volume, which was computed in the previous steps. A potential advantage of this scheme is the possibility of finalizing the computation of the integral before reaching the final location s_n whenever α_{dst} reaches full opacity, which is known as *early ray termination.* We will be using this color compositing scheme in all the visualization algorithms seen along this thesis.

The alternative composition scheme, back-to-front compositing, evaluates the volume rendering equation along a ray coming from the back part of the volume and advancing toward the viewing position. Unlike front-to-back compositing, *early ray termination* cannot be performed with this scheme.

2.5 The Transfer Function

As said, the process of rendering volume datasets relies on performing a *classification* of the volumetric data that allows assigning specific colors and opacities to different data ranges. Through this process, proper classification of the data ranges allows revealing relevant information of the input dataset.

The mapping between the input data values and the output material properties is provided by the so-called Transfer Functions (TF). In this context, a Transfer Function can be understood as a table or a function that maps input data values to output visual properties. The simplest ones are 1D Transfer Functions, and they usually perform a classification of the input data that maps density values to colors and opacities. There also exist more complex Transfer Functions that allow using more input data criteria for the classification, such as the gradient magnitude or the curvature of the scalar field. Of course, the output properties resulting from the mapping can vary from simple RGBA data to more complex optical properties. An in-depth study of the research done on various aspects of TFs is presented in the state-of-the-art paper [50] by Ljung et al. In this thesis, we have always used 1D Transfer Functions. See Figure 2.5 to see an example on how the visualization of the same volume dataset can vary between different Transfer Functions.

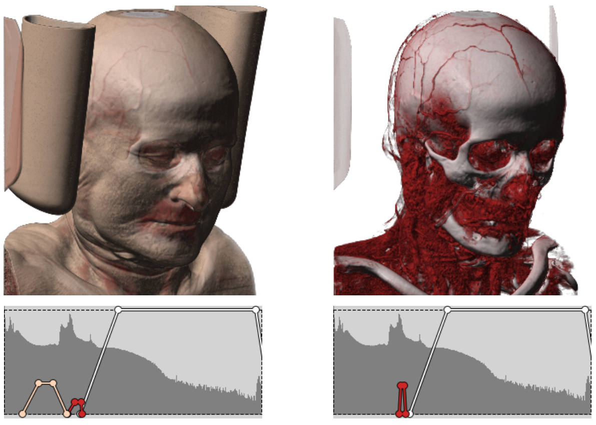

(a) Transfer Function 1 (b) Transfer Function 2

Figure 2.5: The same volume model rendered with two different Transfer Functions. The left image (a) uses a Transfer Function that helps to show the skin tissue with slight transparency. In the right image (b), the piece-wise linear shape in the Transfer Function corresponding to the skin tissue has been interactively removed, so the resulting rendering now exhibits bone and some venous structures.

2.5.1 Pre- and post-classification

As it has been said, rendering algorithms can resample the datasets at random positions in space, an operation that usually involves interpolating the values of samples from the original dataset. Considering this, the mapping performed by the TF can be performed either as a pre-classification or as a post-classification.

Pre-classification performs the mapping assigning the output properties of the TF to the samples of the original dataset (see Figure 2.6-a). This way, the visualization algorithm takes samples at random positions directly interpolating color values after classification. Pre-classification does not reproduce well high frequencies (e.g., sudden peaks in the TF), which generates noticeable artifacts in the final image. Furthermore, pre-classification requires a traversal of the whole input dataset before executing the visualization algorithm whenever the TF changes, which is a bottleneck point in the pipeline if the application is supposed to allow interactive modifications of the TF.

On the other hand, post-classification performs the mapping after the intensity values of the input dataset have been interpolated (see Figure 2.6-b). This way of classification reproduces better the high frequencies in the TF and provides better final image results. Particularly, in models of low resolution, a slightly denser sampling of the scalar field (something smaller than the size of a voxel) can help reproducing those high frequencies, which is not possible at all in scenarios with pre-classified data.

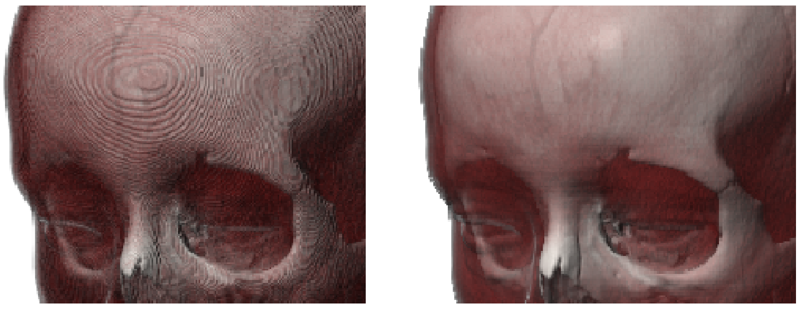

(a) Pre-classification (b) Post-classification

Figure 2.6: Pre-classification (a) consists in applying the Transfer Function in a pre-interpolative stage (i.e. the original voxels are classified). In post-classification, the visualization algorithm takes samples at arbitrary positions, and the reconstructed scalar values are classified after interpolation. As shown, post-classification achieves better results than pre-classification, because applying the Transfer Function after interpolating the voxel values computes more accurate colors for the underlying scalar field.

2.6 Ray casting

Ray casting has been the DVR algorithm used throughout the whole course of this thesis. We have based our work on this method because nowadays it is the most popular image-based method for GPU-aided DVR. Figure 2.7 shows a schematic overview of the ray casting algorithm. The contributing pixels are generated by projecting and rasterizing some *proxy geometry* bounding the volume, such as the minimum axis-aligned box containing the volume. The rasterized fragments are usually mapped to the viewport pixels (if supersampling is not taken into account). Then, for each rasterized fragment, a GPU program following the volume rendering pipeline shown in Figure 2.8 is executed. To start, it casts a single ray that emerges from the virtual camera, passes through the current pixel, and finally intersects the volume dataset. Along the ray, the volume rendering integral (see equation 2.1) is then evaluated by resampling the volume dataset at discrete intervals, usually in front-to-back order. The distance between samples is usually set to some value slightly smaller than the size of a voxel, in order not to lose high frequencies in the scalar field. Throughout the ray traversal, the optical properties of the sampled material are obtained thanks to the Transfer Function and shaded using the scalar field gradient at that position. Finally, the color of each pixel is obtained by iteratively compositing the colors obtained from each sample over the ray (equation 2.3).

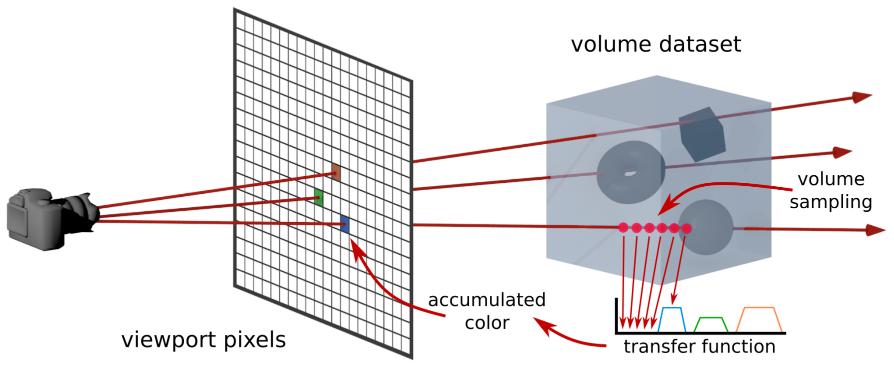


Figure 2.7: In the ray casting algorithm, the color for each pixel is computed by color-compositing the optical properties of the densities (which are obtained from a Transfer Function) as resampled from the volume dataset.

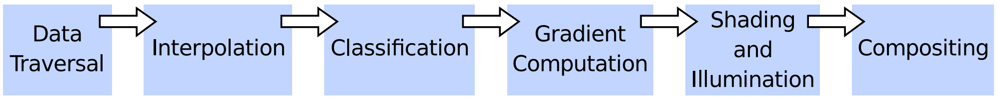


Figure 2.8: The volume rendering pipeline. In order to evaluate the resulting color of a pixel, the dataset must be traversed, taking samples at arbitrary positions. Values at those samples are evaluated by means of an interpolation filter applied on the discrete dataset. Density values are in turn classified with a Transfer Function that provides their optical properties. The gradient can be also estimated for each sample position in order to apply an illumination model. In the last place, the evaluated colors of all samples in the traversal are composed along the viewing rays to obtain the final image.

2.6.1 The volume rendering pipeline

The volume rendering pipeline for the ray casting algorithm, like for any other DVR method, has several subsequent stages (see Figure 2.8):

- **Data Traversal.** The input dataset is resampled at discrete positions in order to evaluate the volume rendering integral.
- **Interpolation.** Usually, the sampling positions do not match the exact locations of the grid points. Different filters can be used in order to reconstruct the continuous 3D scalar field at those sampling positions, being tri-linear interpolation the most common for uniform grids (and also the one implemented by common graphics hardware).
- **Classification.** Classification maps values of the input scalar field to optical properties, which allows distinguishing different materials in the volume. This feature is usually provided by Transfer Functions, which are basically tables that assign the optical properties in the form of color and opacity.
- **Gradient Computation.** When local illumination is a desirable feature, the gradient of the scalar field is typically needed. Gradients are usually approximated at the sampling positions of the data traversal with gradient estimation techniques such as central differences.
- **Shading and Illumination.** Volume shading can be incorporated by adding an illumination term to the emissive source term in the volume rendering integral. This illumination term is computed using the computed gradient, and the emissive term is obtained by the classification made with the Transfer Function.

- **Compositing.** The iterative computation of the discretized volume rendering integral (equation 2.2) leads to compositing schemes that allow computing the final color incrementally as the data traversal occurs, either if it happens front-to-back (equation 2.3) or back-to-front.

2.6.2 Improvement techniques

The contributions presented throughout the course of this document, unless otherwise stated, will assume the usage of some implementation details that either boost performance or improve the quality of the final renderings. They are briefly introduced in the subsequent paragraphs.

Empty space skipping (ESS)

Whenever some of the structures of the volumetric dataset are not required for the final rendering, the Transfer Function sets the opacities for these densities to zero. When this happens, the sampling taken over the ray traversal is wasting time and computational power, compromising the performance of the process. ESS is a performance improvement that avoids sampling over those transparent ray segments that will not have an impact on the final color. We use a basic form of ESS [44] that consists in subdividing the proxy geometry (which is originally a cube-like bounding box) into smaller blocks (see Figure 2.9). These smaller blocks are classified in a pre-processing step, and they contain the minimum and maximum scalar values contained within its volume. Based on the Transfer Function and the minimum and maximum values contained in these blocks, we can quickly determine which blocks will be visible or not. All those visible blocks will build a tightener proxy geometry that will generate fewer fragments in the viewport space and will also determine a shorter integration domain over the ray.

Early ray termination (ERT)

Another improvement related to performance is ERT. When tracing rays across a volume dataset in a front-to-back fashion, many rays will quickly accumulate full opacity as color compositing occurs. It makes no sense to continue iterating through the rays if the contribution of the subsequent samples will not contribute color and have an impact on the final rendering. ERT [44] is

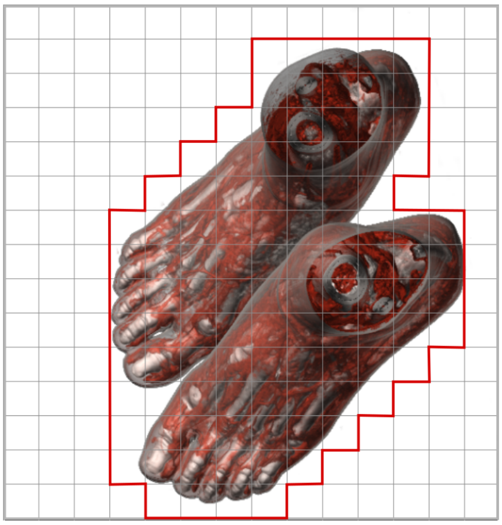
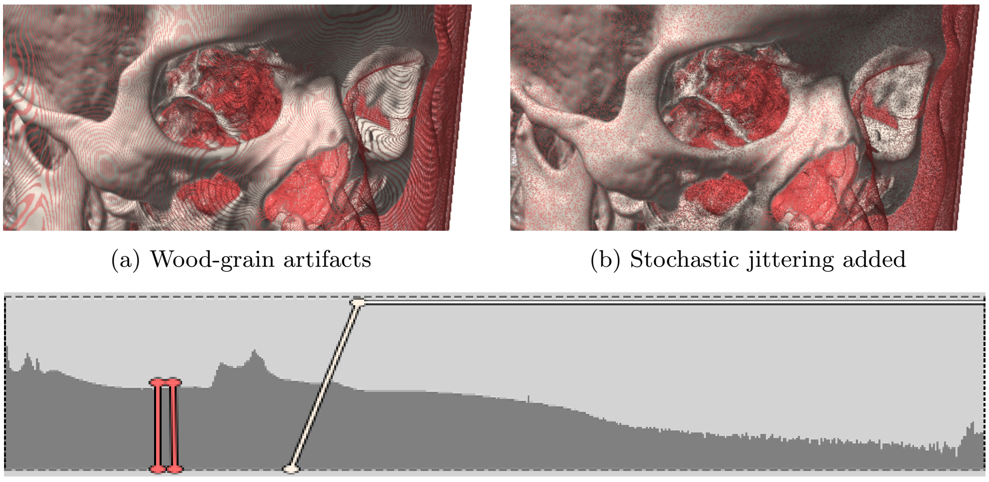


Figure 2.9: The proxy geometry bounding this volume model is shown in red (in 2D). We subdivide the bounding box of the volume model in a grid and generate a mesh that envelopes those grid cells containing non-transparent data. We use the proxy geometry in order to perform empty space skipping, allowing rays to effectively start where non-transparent data is found, and finishing wherever there is only transparent data remaining.

another performance improvement that consists in finalizing the ray traversal at this point. Thanks to the dynamic branching provided by nowadays' GPUs, this operation is easily performed in fragment shaders, where a conditional statement is enough to stop the execution of a loop.

Stochastic jittering

One of the drawbacks resulting from resampling a volume dataset at discrete intervals in order to evaluate the volume rendering integral is the potential loss of high-frequencies both in the scalar field and in the Transfer Function, which generates the well-known *wood-grain* artifacts (see Figure 2.10-a). This issue particularly happens when the sampling rate is too low, and it may be more easily noticed when using Transfer Functions containing peaks with a steep slope. It can also be seen in regions of sudden change in the scalar field such as in the boundary between air and skin tissue. In order to capture these high frequencies, an increase of the sampling rate over the ray could be added. However, this solution directly decreases performance. This is not acceptable for our purposes, as it will be shown in the following chapters, where we will


(a) Wood-grain artifacts

(b) Stochastic jittering added

(c) Transfer Function used for the above renderings

Figure 2.10: With an insufficient sampling rate, high frequencies either in the Transfer Function (c) or in the scalar filed are sometimes missed. As shown in (a), this may result in *wood-grain* artifacts due to the spatial coherency among neighboring pixels. With stochastic jittering (b), these manifested artifacts are suppressed by noise.

deal with multiresolution approaches which will also be combined with lower sampling rates in order to increase performance.

One common solution to remove these artifacts without increasing the sampling rate is stochastic jittering. This technique hides *wood-grain* artifacts by adding small offsets to the sampling positions of rays in the viewing direction. The sampling positions along each ray through the pixel are offset by a different random factor. Consequently, the coherence between pixels that manifests as artifacts is suppressed by noise. The implementation of this technique is simple and only requires a simple 2D texture with random noise. Each pixel in the viewport is mapped to a texel of this noise texture, and the resulting ray passing through this pixel is accordingly offset by the fetched random value from the texture. The results of using this technique as opposed to visualizations with *wood-grain* artifacts can be seen in Figure 2.10-b.

Pre-integrated volume rendering

In order to solve the previously mentioned *wood-grain* artifacts, instead of increasing the sampling rate, or using stochastic jittering (see Section 2.6.2), the solution by Engel et al.[23] pre-computes the integration between two different

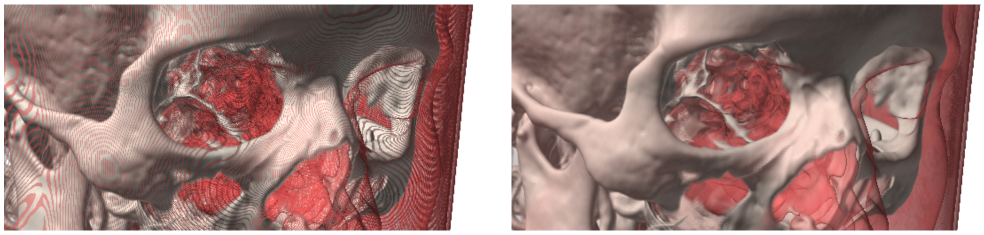

(a) Wood-grain artifacts (b) Pre-integrated volume rendering

Figure 2.11: Using standard 1D TFs, when the size of the sampling step is not small enough, high frequencies either in the TF or in the scalar field can be missed, showing these *wood-grain* artifacts (a). With pre-integrated volume rendering [23], this issue is solved (b) without compromising performance.

density values of the scalar field for a certain separation between samples. In order to do this, the sampling rate (separation between samples) for a certain dataset and for each of its coarser dataset representations (in multiresolution models) is fixed, and the pre-integrated Transfer Functions for each one is pre-calculated in a separated table. These tables are actually 1D SATs (summed area tables) that store, at each entry, the summation of the values of the original TF from the beginning of the table to the current entry. With them, the average contribution of colors and opacities in the interval between two samples over the ray can be estimated. Figure 2.11 shows the results achieved using pre-integrated volume rendering as opposed to using standard TFs.

Note that stochastic jittering (Section 2.6.2) removes *wood-grain* artifacts in favor of noise, which can be annoying in certain circumstances. Pre-integrated volume rendering, on the other hand, achieves much softer results, but it does not fully remove *wood-grain* artifacts because it assumes linear changes in the density between two samples, which is not the case in many occasions. The implementations of the visualization algorithms used in this thesis use both, the combination of stochastic jittering and pre-integrated volume rendering, which highly improve the quality of the final renderings.

2.7 Gradient-based illumination

Traditional local illumination models use the notion of the normal vector, which describes the local orientation of a surface, to compute the local illumination. In DVR, there is not a single surface, defined by a discrete mesh, for which normals can be provided at each vertex. Instead, we assume that light is reflected at *isosurfaces* inside the volume data. For a fixed *isovalue* v, an *isosurface*, in volumetric models, is the union of all sample points p that share the same scalar value v. The scalar field gradient for an *isosurface* can be computed at a certain point of the volume with the derivative of the scalar field:

$$\nabla f(x) = \begin{pmatrix} \frac{\delta f(x)}{\delta x} \\ \frac{\delta f(x)}{\delta y} \\ \frac{\delta f(x)}{\delta z} \end{pmatrix} \tag{2.4}$$

which points into the direction of steepest ascent, which is always perpendicular to the *isosurface.*

There are several methods for estimating gradient vectors, which differ in the accuracy of the resulting gradients and the computational cost. Some of the most common methods are based in *finite differences* (e.g. *forward differences* or *central differences*), which are generally cheap options that obtain quite acceptable results, or in *discrete filter kernels*, such as the *Sobel operator*, which achieves better results at the expense of a higher computational cost. The decision on which gradient estimator to use is mainly tied to the availability of time at the stage of computation.

2.7.1 Pre-computed vs. on-the-fly gradients

There is a variety of techniques that allow us to estimate the gradient from discrete data. In GPU-based DVR, the two most important ways of obtaining gradients while the visualization algorithm takes place are either having pre-computed gradients at the discrete positions of the samples of the original dataset, or by estimating them on-the-fly, which means that the computation of the gradient needs to be executed in real time.

The estimation of pre-computed gradients takes place in a pre-interpolative stage. Gradients are evaluated either from the scalar field, or from voxels' opacities if working with pre-classified data. In either case, the evaluation will take place at the location of the original samples of the regular grid. This implies that the visualization algorithm will obtain interpolated gradients, which will be fast to fetch, but whose values will not be the same as if estimated directly from the interpolated scalar values. Pre-computed gradients may be estimated with higher quality filters that take longer to compute, as time restrictions are more flexible at this stage.

On the other hand, gradients evaluated on the fly in shader code take place after interpolation has occurred, which allows evaluating gradients directly at arbitrary positions between the original samples of the regular grid. This is obviously slower than fetching a pre-computed gradient, but the evaluated gradient will be more accurate with respect to the interpolated scalar values. However, if gradients need to be estimated at this stage, faster approaches are preferred in order not to compromise factors such as interactivity or power consumption.

2.7.2 Issues related to gradient illumination

In practice, several problems occur when gradients are used in order to compute local illumination. The first one is when trying to estimate the gradient in a homogeneous area with no data variation. In such case, there is not a well defined *isosurface*, and thus, the estimated gradient may tend to be $(0, 0, 0)$, or any other tiny value with an undefined direction due to numerical inaccuracies or small noise in the dataset. Another issue related to multiresolution datasets is that, as a consequence of the loss of information, *isosurfaces* have a different topology in coarse datasets, and thus, gradients are also different, providing an inconsistent shading among levels of resolution.

2.8 Visualization artifacts and performance issues

As we have seen, the proposed visualization pipeline consists of several stages where the involved dataset suffers transformations. From acquisition to the final rendering, the dataset can be filtered, downsampled, and rendered in different ways. These transformations may have an impact on the accuracy of the final ray casting images, and may also require an important amount of computational resources.

In order to develop good quality and efficient solutions for the visualization of medical models in mobile and commodity devices, in this thesis, we propose improvements to existing flaws in those stages of the volume rendering pipeline. These solutions and improvements can have either impact in the results regarding two different aspects: visual quality and performance/interactivity.

2.8.1 Visual quality

With the aim of properly managing volume datasets on mobile devices, we make use of multiresolution techniques, where coarse representations of the original dataset have smaller resolutions, and thus, loss of information, which directly affects the fidelity and the visual quality of the generated renderings. We have identified the following issues, regarding visual quality:

- **Loss of features in downsampling:** With the aim of providing high frame rates to provide users interactive camera movements, we will be working with a multiresolution scheme, which involves filtering and downsampling strategies (see Figure 2.1). This is the most aggressive data transformation where a lot of information (e.g., fine details) is lost in the coarse dataset representations after downsampling. Figure 2.12 shows an example of this artifact. Chapter 4 introduces a novel downsampling filter to preserve features which are prone to disappear in this process.

- **Erroneous shading in coarse datasets:** Many implementations of the ray casting algorithm compute gradients on-the-fly directly in the fragment shader, where the ray traversal is performed. Computing gradients this way from a coarse dataset that has been previously downsampled, does not provide gradients that match the directions of the gradients in the original dataset in many cases. Figure 2.13 shows this issue. The shading of surfaces is a significant contribution to the quality of the final renderings, and it is very sensitive to the quality of the gradients provided. To improve this, a proposal that pre-computes gradients from the original dataset and applies an improved downsampling filter, along with an efficient encoding scheme, is explained in Chapter 5

- **Incorrect colors and opacities in coarse datasets:** A Transfer Function is usually designed to show specific properties of a dataset. They can be used to visualize similar datasets, such as in medicine, but slights adjustments are necessary in order to have the best results. As it is shown in Figure 2.14, using a TF that was specifically designed to visualize a certain dataset will not be the optimal solution to visualize its coarse representations in the multiresolution hierarchy, because the distribution of values varies among levels of resolution. To address this issue, in Chapter 6 we present an approach that adjusts a TF originally designed to visualize a given dataset, and creates adjusted TFs for its coarser representations.

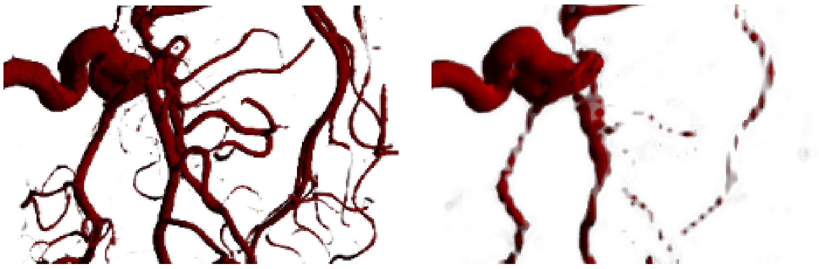

(a) Full resolution (b) Low resolution

Figure 2.12: Loss of features in downsampling. The renderings of the full resolution model (a) have venous structures that mostly disappear in the rendering of the lower resolution version of the dataset (b).

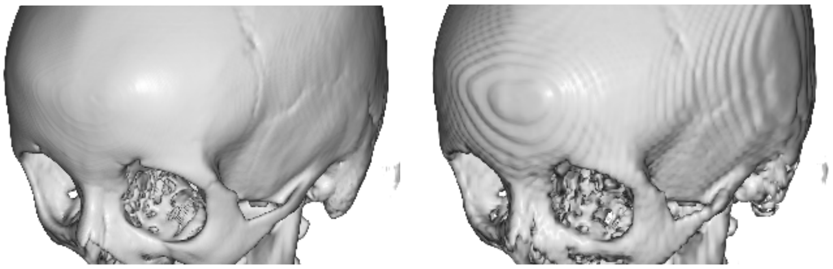

(a) Full resolution (b) Low resolution

Figure 2.13: Inaccurate shading after downsampling. Images generated from the high-resolution dataset (a) show a smooth surface on the skull. After downsampling (b) the surface exhibits a noticeable staircase artifact mainly due to the shading calculations.

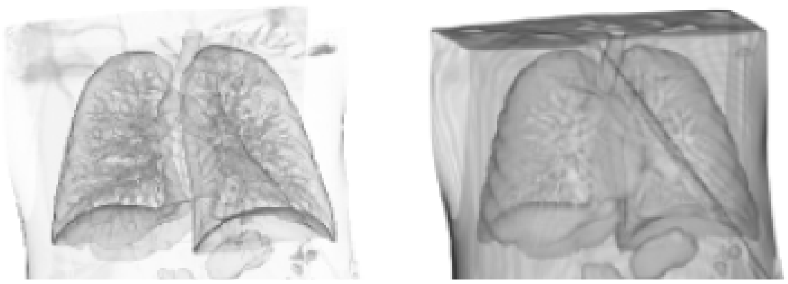

(a) Full resolution (b) Low resolution

Figure 2.14: Incorrect colors and opacities in coarse datasets. During downsampling, some tissues, if not disappear, become thicker due to the loss of resolution. If these tissues are assigned a semi-transparent color by the Transfer Function, this thickening effect affects the final color of the resulting output image. The full resolution model (a) exhibits a lower level of transparency than the low resolution image (b), because the skin tissue is thicker in the second one.

2.8.2 Performance and interactivity

In medical applications, rendering the full resolution datasets would be desirable, but it is not always possible due to hardware limitations. This kind of software at least needs to give the possibility to generate high-resolution images at some point. However, the generation of high-resolution images mainly leads to performance issues:

- **Loss of interactivity:** Mobile devices, tablets, and commodity PCs are the hardware which is most available in clinics and hospitals. Those devices are not provided with high-end graphics chips, so generating renderings of high-resolution datasets in high-resolution viewports takes a considerable amount of time. In the case of mobile devices, a high-resolution render can take seconds in complete, which provides blocking, non-interactive manipulations, and in the worst case, depending on the device, application crashes. In Chapter 7, we address this problem with multiresolution datasets and progressive render algorithms. Whenever interaction is necessary, coarse datasets are rendered using the quality improvement contributions presented in the previous chapters, and then, at still frames, a progressive ray casting algorithm renders a higher resolution dataset incrementally.

3 State of the Art

This chapter reviews the state of the art in the field of *Volume Visualization* that is most related to our global goal: the interactive, high-quality visualization of big datasets in commodity and mobile devices. To that aim, the following sections develop upon our areas of interest in the field. First, Section 3.1 presents an overview of contributions or approaches related to multiresolution algorithms, dataset compression, and downsampling methods. Section 3.4 reviews literature that strives to improve the quality of multiresolution volume visualizations, and finally, Section 3.5 cites the most recent publications tackling the problem of interactive direct volume rendering on mobile devices.

3.1 Multiresolution volume datasets

With the improvement of modern acquisition devices, the resolution of medical image datasets has increased continuously. The amount of memory of recent GPUs is also growing, but unfortunately, the increasing rate of the size of volumetric datasets is even much higher. Different rendering techniques based on multiresolution have been proposed to obtain interactive visualizations. These techniques rely on the generation of several levels of detail from the original dataset and using coarse levels of resolution for the visualization task when convenient, in order to find an appropriate balance between image quality and interactivity.

Several multiresolution techniques use special data structures such as octrees [7, 46, 29], N^3-trees [11], 3D mipmaps, hierarchical grids [26], or other sparse representations [77, 89], in some cases also combined with compression techniques [28, 24] and streaming [80]. Some techniques approach the multiresolution issue through wavelets [31]. In [32], the authors present an algorithm for the rendering or large volume data based on a compressed hierarchical wavelet representation of the input dataset. In [82], the authors present a parallel multiresolution volume rendering framework for large-scale time-varying data visualization using wavelet-based time-space partitioning (WTSP) trees. We will concentrate on regular grids, as they are the most commonly provided by capture devices in the medical practice.

Some authors have focused their efforts to solve common artifacts regarding mixed multiresolution visualizations. For instance, in [87], an adaptive approach to volume rendering via 3D textures where their level-of-detail representation guarantee consistent interpolation between different resolution levels is presented. In [5], the authors propose a mixed-resolution volume ray casting approach that allows freely mixing volume bricks of different levels of resolution during rendering. Their framework and packing scheme allows mixing the different bricks in a single 3D texture and is able to obtain a C0-continuous function over the whole dataset with hardware-native filtering.

Frequently combined with multiresolution approaches, many articles focus on lossy or lossless data compression in order to reduce the amount of volume data. For more in-depth information about compressed GPU-based DVR, the survey on this subject by Balsa *et al.* is a good starting point [2]. Another group of techniques partition data in order to make smaller chunks that are then loaded on demand to the GPU. These techniques include bricking, streaming, and complex algorithms to transmit data from the main memory to the GPU memory efficiently. Beyer *et al.* have gathered quite recent related works in their survey [6].

Not directly related to multiresolution, but definitely, an important issue to deal with is the processing of the volumetric datasets. The quality of the values obtained from datasets represented by uniform grids of samples can be improved by pre-filtering or using adequate reconstruction filters. For instance, in [60], the authors present a feature-preserving volume filtering method based on the optimization of a three-component (original value, feature preservation, and curvature minimization) penalty function. The solution they propose can be efficiently solved in the frequency domain using fast Fourier transformation (FFT). With this filter, typical staircase artifacts are eliminated from visualizations without losing fine details. Another, more recent approach, is the pre-filtered Gaussian reconstruction scheme presented in [12]. This reconstruction filter first performs a Gaussian deconvolution in the frequency domain as a pre-process, and then, a spatial-domain convolution with a truncated Gaussian kernel. Although not yet implemented in conventional graphics hardware, they approximate an ideal reconstruction of the signal and achieve results significantly better than previous reconstruction approaches.

In this thesis, we are focusing on generation techniques for multiresolution models that require little pre-processing time and computational complexity in general. For this reason, recalling that a desirable requisite is that the processed medical data is available for its inspection as quickly as possible, we have strived to avoid using compression techniques or expensive filters that hinder our performance goals.

3.2 Downsampling of scalar data

An important step in all multiresolution approaches is the way in which reduced versions of the original data in the multiresolution structure are obtained. The better the downsampled versions, the better the final images that will be obtained after rendering.

Following this line, downsampling by means of averaging voxels, or possibly filtering and subsampling the original dataset, are the most common approaches [7, 49]. Unfortunately, these techniques may rapidly eliminate small details, as it is expected when the resolution decreases.

In [84], the authors present a feature-preserving volume data reduction and focus+context visualization method based on Transfer-Function-driven, continuous voxel repositioning and resampling techniques. Their method uses a pre-defined Transfer Function in order to perform importance sampling so that the interesting regions are captured with more detail. Nevertheless, using Transfer Function dependent techniques is not appropriate in scenarios such as the presented in this thesis, where medical data can be visualized using Transfer Functions that can be potentially changed.

Another data reduction technique is topology-guided downsampling [43], a downsampling method for structured grids which preserves much more of the topology of a scalar field (i.e., the number of components, tunnels, holes, etc.)

than other existing downsampling methods by preferably selecting scalar values of critical points. Unlike the previous method, it is completely independent of the Transfer Function, only taking into account the topology of the original scalar field. However, the spatial bias introduced by this topology preserving technique is not desirable in medical visualizations, as the location of features in the downsampled models changes with respect to the original dataset.

There is another family of methods tailored to improve the quality of the volumetric datasets by reducing noise. These are more suitable for datasets with a high amount of noise, such as the ones from ultrasound imaging. In general, such techniques are more focused on noise reduction, but sometimes they also combine the noise reduction with downsampling. Kwon et al. [45], for instance, filter and denoise ultrasound images using a fast bilateral filter. As it will be reviewed in Chapter 4, the bilateral filter usually produces undesirable results for downsampling medical datasets such as the ones obtained from CTs. Another technique, more focused on noise reduction, is the method by Wang et al. [83].

There is also related bibliography about feature preserving downsampling methods for 2D images. Although these methods have not been explicitly designed for volumetric datasets, their idea could be easily adapted to such case. For instance, Kopf et al. [40] implement a content-adaptive downsampling method that uses a bilateral combination of two Gaussian kernels defined over space and color, calculated using an iterative maximum-likelihood optimization process using a variation of the Expectation-Maximization algorithm. Öztireli and Gross [68], formulate image downsampling as an optimization problem that maximizes a perceptual image quality metric (SSIM) based on the difference between the input and the output images. The authors state that their method is preferred by most people, and its performance is faster than the former. Anyway, with these methods, the process of downsampling small 2D images takes a computation time of the order of seconds or minutes, which translated to volumetric datasets in medicine would result in much longer times (at least hours). Furthermore, Weber et al. [86] present an algorithm based on convolutional filters where input pixels contribute more to the output image the more their color deviates from their local neighborhood. They use a guidance image based on a preliminary downsampled image to be able to define pixel contributions, and a parameter λ that has to be manually adjusted, depending on the image, to provide the best results. This filter, although presented for 2D images, is very similar in spirit to the filter we present in Chapter 4. However, they need a manual adjustment of the parameters.

In Chapter 4 we propose an alternative solution for downsampling scalar data. We present a downsampling filter [15, 17] designed to preserve features typically lost during the generation of coarse representations of volume datasets, and still obtaining smooth results. The presented method takes just a few minutes to generate a downsampled version from any of the high-resolution datasets we used. Furthermore, and not less important, it runs completely unattended, so no need for any user configuration or parameter adjustment is necessary.

3.3 Downsampling of gradient data

An important stage in the volume rendering pipeline is the computation of gradients, which are used to simulate both diffuse and specular reflections (in our case, using the Phong shading model [71]). Therefore, if its calculation is not accurate, the quality of the resulting rendered images may be affected as mentioned in Chapter 2, and also shown in Chapter 5. In [4], the authors present an analysis of the ideal gradient estimator. There are several methods to evaluate gradients from a scalar field. One of the most used reconstruction filters designed for that purpose is the central differences approach [34], which requires six extra texture lookups to perform the difference in the scalar field along each direction in the XYZ space. There is an even faster version of this filter, at the expense of introducing a small spatial bias, that only uses three extra texture lookups by calculating the difference with the central density. There are also methods that achieve gradients of better quality such as the Sobel's operator [20] by sacrificing the performance. The classical Sobel's operator, for instance, requires 26 extra texture lookups due to its $3 \times 3 \times 3$ kernel. To alleviate this performance penalty, Sigg and Hadwiger [70], use a more efficient version of the Sobel's operator that only needs eight extra texture lookups at the corners of the voxel containing the sample to shade, and still obtain similar quality results. Another gradient estimation approach for volume data (which also provides filtered densities) based on 4D linear regression is presented in [59]. Their solution leads to a system of linear equations that can be solved with an efficient convolution, which takes 26 accesses to the neighboring voxels, similar in spirit to the Sobel operator.

Working with pre-computed gradients [34] allows using slower but accurate computations for estimating gradients before rendering (as during pre-process, speed is usually not crucial) and speeds up the visualization algorithms taking place in the GPU by moving this rather expensive computation to previous stages. However, most existing algorithms for encoding normal vectors and gradients cannot be used in the context of volume rendering with gradients encoded in a 3D texture. Gradient values obtained between several voxels are always the result of a convex interpolation when the gradients in voxels are simply encoded by quantizing their cartesian components (G_x, G_y, G_z). Unfortunately, this desirable property is not fulfilled by many other well-known proposals like [13] (spherical coordinates), [66] (recursive subdivision of a Platonic solid), [14] (indexing spherical triangles) or [9] (encoding a point in the surface of a cube).

In Chapter 5 we propose a solution based on pre-computed gradients. Our approach consists of a downsampling filter to generate multiresolution representations of gradient data and an encoding scheme based on a monotonic transformation that guarantees the property of interpolation mentioned above, thus ensuring that the final algorithm is GPU-friendly.

3.4 Quality visualization of downsampled data

The previous sections have dealt with the generation of downsampled models that preserve quality. However, we can obtain higher quality renderings by using other techniques that can complement the previous approaches.

Younesy *et al.* [90] focus on improving the quality of renderings for coarse multiresolution levels. They state that the original data distribution in coarse models might be ideally approximated by storing local histograms at each low-resolution voxel. However, as the authors note, this is usually impractical due to its high storage requirements. Thus, they propose a simplification that consists of representing these histograms with a Gaussian basis function, which implies storing an average density (μ) and its standard deviation (σ), along with each voxel. Although they designed an efficient algorithm, the size of the data is increased with respect to traditional downsampling methods, and this may be a problem if the available memory is limited.

In [42], Kraus and Bürger defend the fact that downsampling of RGBA data is a better approach to compute multiresolution hierarchies rather than downsampling scalar grids, because mixing colors during downsampling produces the expected results, whereas averaging densities can provide inconsistent colors after post-classification, depending on the Transfer Function and the nature of the data itself. In their paper, the authors present a sampling method for RGBA volume data that can be also used in the context of multiresolution datasets. They demonstrate that their method is applicable to the construction of multiresolution hierarchies of RGBA volume data such as mipmap volume textures. Although GPU ray casting is typically performed over scalar data in many applications (thanks to the ease it provides in order to have interactive changes of the Transfer Function), they state that actual modifications of the Transfer Function for pre-classified data in interactive time are practically possible with modern graphics hardware. However, in many cases, the preferred option is still storing the scalar field of densities into a 3D texture. This avoids increasing $\times 4$ the memory requirements (as RGBA textures do) which is a scarce resource in commodity PCs and mobile devices, and allows performing post-classification during the ray traversal in the ray-casting algorithm, where the density values can be mapped to opacity-weighted colors by means of a Transfer Function that can be modified interactively.

More recently Sicat *et al.* [77] have presented an approach that uses a compact sparse representation of probability density functions (pdf) to capture voxel neighborhood distributions for consistent multiresolution volume rendering. They succeeded in avoiding erroneous data analysis (loss of information) when coarser models are rendered using the same TF than the initial volume. However, the significant pre-computation time needed and the increase of storage makes this representation impractical in the current clinical practice. Our objective is to obtain consistent visualizations for datasets commonly used in medical environments, minimizing the modifications needed to the rendering pipeline, the preprocessing time, and the increment of memory storage.

In this line, in Chapter 6 we present a method to improve the visualization of multiresolution datasets by means of adapting the original TF for coarse levels. The solution we propose does not require such extra storage but a simpler small Transfer Function mapping. Furthermore, our system has no impact in terms of required computational power since we use the same ray casting algorithm with no modifications.

3.5 Visualization techniques on mobile devices

Since mobile platforms are ubiquitous nowadays, the interest in using mobile devices for rendering volumetric models, especially medical datasets, is growing. Approaching these techniques to ubiquitous devices has benefits in several fields of the medical practice such as diagnosis, treatment, or teaching [62] purposes. Some papers have been published compiling previous work regarding this field of visualization. In [74], a state of the art of mobile rendering for iOS devices is presented. In [63], the authors present a wider analysis of several publications in this field including all kinds of mobile devices.

Several visualization frameworks [79] and applications have been developed to allow interactive visualizations on mobile phones. Focusing on the rendering of volume models, an example of a web-based DICOM viewer is Oviyam [72], an HTML 5 solution that allows displaying series of studies as JPEG images. OsiriX [67] is a widely used DICOM viewer which is prepared to perform interactive direct volume rendering on mobile devices. Another toolkit which allows such kind of visualizations is VES[39], a VTK OpenGL rendering kit also prepared for mobile devices.

Many previous approaches have addressed the visualization of volumetric models on mobile devices using two strategies: server dependent methods and local methods. The following sections provide references to the research carried out in both strategies.

3.5.1 Server dependent methods

Server dependent methods rely on external servers in order to perform part of (or the whole) the work of the visualization pipeline. Depending on the level of dependency, these methods can be classified in *thin*, *balanced*, and *fat* approaches, depending on their dependence [63].

Server dependent methods that heavily rely on the server side to perform all power consuming operations are called *thin client* architectures. Following this scheme, Lamberti et al. [47] communicate rotation and translation commands from client devices to the server, and obtain an MPEG video stream with the rendered results of medical images as a response. In [33], Hachaj et al. propose a similar solution also based on *thin clients*, and Gutenko et al. [30] use a more efficient and modern video codec (H264) to encode the video stream. However, these methods have strong connectivity restrictions we want to avoid.

Balanced solutions distribute the tasks between the server side and the mobile device. In this line, Campoalegre et al. [8] perform a block-based Transfer Function aware compression of the target dataset and are able to transmit the desired regions of interest to support adaptive ray casting on the client side.

Fat distribution schemes take more advantage of client desktop machines and hand-held devices [63]. They mainly rely on the server to provide the datasets after possibly performing some expensive pre-processing tasks, but produce the render locally. For instance, Congote et al. [10] present a platform implementing this kind of architecture by means of the WebGL standard. Movania et al. also developed various algorithms that perform a single-pass ray casting for the efficient visualization of medical models based on WebGL. For instance, in [57] and [55] they present a single-pass volume rendering algorithm for 3D medical images using OpenGL ES 2.0. Furthermore, they present another two high-performance volume renderers in [54]. In addition, the same authors explain an algorithm that allows performing real-time volumetric lighting on WebGL platforms in [58]. Using a different approach, a details-on-demand scheme is presented by Schultz et al. [75], where they allow the user to explore the entire dataset at its original resolution while simultaneously constraining the 3D texture size so that it does not exceed the GPU capabilities of the portable device.

3.5.2 Local methods

Local rendering methods allow the visualization on mobile devices with no need of network connectivity. 3D textures have been widely available on mobile GPUs just recently, so most methods for rendering volumetric models have relied on 2D texture stacks or tiled 2D textures emulating 3D textures. Among others, Moser and Weiskopf [53], Fogal et al. [25] and Noon et al. [65] have developed tools using stacks of 2D textures representing the 3D volume. Congote et al. [10], Noguera et al. [64, 61] and Movania et al [56], on the other hand, emulate 3D textures by using a mosaic layout of its slices within a set of 2D textures. More recently, when 3D textures have become widely available, both slicing and ray casting algorithms have been used. In [3] Balsa et al. presented a practical comparison of volume rendering using several devices and algorithms, including ray casting with the use of 3D textures, which was far from interactive at that time. Also using 3D textures, Xin and Wong [88], presented an intuitive framework for volume data exploration, although they don't work with datasets of resolutions higher than 128^3.

Nowadays, GPUs in hand-held devices are more capable, so focusing on *fat* and local rendering approaches by implementing the ray casting task on mobile phones seems more feasible. However, porting volume rendering to mobile devices may be challenged by three main limitations: GPU capabilities (as they could not provide the proper features to deal with the algorithms used to visualize volumetric models), RAM size (models might not fit in main memory), and GPU horsepower (even though models might fit the GPU memory, the frame rate achieved could be inefficient to support interactivity adequately).

3.5.3 Progressive methods

Although not explicitly working on mobile platforms, the following publications target the issue of fast volume rendering through incremental ray casting algorithms. Levoy [48] introduced an incremental way of performing volume ray casting based on an adaptive image space subdivision. In [41], Kratz et al. improved Levoy's approach by introducing an error estimator from the field of finite element methods. In the same line, Frey et al. [27] presented a scheme for progressive rendering that adapts to different changes during data exploration. They demonstrate an automatic parameter optimization scheme using a video metric to optimize their frame control. These techniques are mainly focusing on quality metrics to lead the progressive refinement algorithm. However, the way they distribute rays is not tailored to achieve optimal performance, which is a very important factor on mobile devices.

To tackle the problem of interactive quality volume visualization on mobile devices, in Chapter 7 we present a framework for the multiresolution visualization of volume datasets that uses several techniques based on incremental rendering, and obtains high-quality results without sacrificing interactivity.

4 Downsampling of Scalar Fields

After data acquisition, the next stage of the visualization pipeline is data processing. At this point, the generation of multiresolution hierarchies is a crucial step to achieve interactive visualizations of large voxel models. Through the downsampling of large volumetric models, we generate coarser models that fit the limited capabilities of commodity and mobile devices. However, the loss of resolution comes with a loss of fine details. Regarding that, in this chapter, we present a comparison of several existing downsampling methods for volumetric models and analyze their benefits and shortcomings. Based on the observations made, we finally present a novel feature-preserving downsampling filter for volumetric datasets.

4.1 Motivation

Interactive visualization of large scalar grids is a required task in fields such as medical imaging. Several operations, such as model inspection from different points of view or modification of the Transfer Function to reveal specific features, are the typical operations that physicians need to perform on a regular basis. Interactive visualizations of the volumetric datasets are necessary to allow these actions. However, datasets of high resolutions may hinder interactivity, especially when visualization algorithms are required to run in modest hardware, or even in mobile devices.

A standard approach to provide a higher degree of interactivity when dealing with large models is to reduce the amount of volumetric scalar data in order to obtain coarse, simplified representations that are easier to handle with the available hardware resources. This process is typically known as *downsampling*, because it basically consists in reducing the number of samples of the original dataset. This data reduction process takes place typically in a previous stage as a pre-process, before the visualization task is required. As said in Chapter 2, we will focus on the generation of multiresolution hierarchies of voxel models, which are the most efficient representations for GPU-aided volume rendering, and the most commonly used by commercial applications in the medical environment.

However, one of the common shortcomings of most downsampling methods is that an important amount of information is lost, resulting in images of lower quality when rendering the coarse, downsampled models, which tend to disrespect the original topology and lack fine features of the original dataset.

4.2 Problem addressed

This chapter addresses the problem of information loss during the downsampling process. Naive downsampling methods such as plain subsampling, averaging or Gaussian filtering, are widely used in many volume visualization applications. These methods are easy to implement and have cheap requirements in terms of memory resources and computation time, but the resulting quality of the downsampled data is clearly reduced. To alleviate this, proper filtering of the original dataset is an important procedure to carry out before subsampling. We have performed an analysis of the typical filters used for downsampling datasets. Based on the observations made on their behavior, and in order to reduce the amount of interesting information lost due to downsampling, we propose a novel downsampling filter based on a Gaussian filter that is able to preserve structures that are prone to disappear with standard methods. The method we propose is simple, easy to implement, fast, and provides good quality results with preservation of features.

4.3 Analysis of existing downsampling techniques

We have analyzed several downsampling methods for volumetric datasets that range from the typical pure subsampling (i.e. taking one of the samples from the original dataset) to more elaborated approaches such as the topology-guided downsampling by Kraus and Ertl [43]. We have observed that the most important problem with most techniques is the lack of preservation of fine details. For completeness, we also experimented with other possibilities such as combining any of the previous downsampling methods with an extra previous noise reduction stage. Unfortunately, no significant gains are obtained. Thus, for the sake of clarity, we only comment the tested methods that exhibit a noteworthy behavior and analyze the reasons behind their results. It should be observed that we are not performing an intrinsic analysis of these downsampling techniques, but an analysis of the resulting visual quality when they are used in the context of ray casting volume rendering and 1D Transfer Functions.

4.3.1 Testing conditions

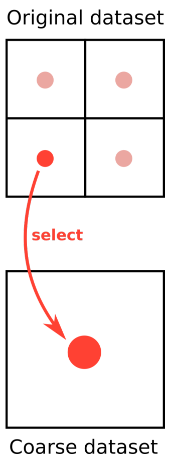
The results shown in the figures of this chapter have been obtained using a GPU ray casting algorithm, with a distance between samples proportional to the size of a voxel (see Section 2.6), and pre-integrated classification. The camera has been configured to use an orthographic projection of the scene, as it is usually required to visualize medical images. The values sampled from each ray have been colored with a pre-defined Transfer Function, and shaded with the Phong reflection model [71], using the gradient computed on-the-fly at each sampling step. We have used three datasets, each one with a specific Transfer Function, designed to reveal specific features of each model:

- **Ribs dataset:** Original resolution (512^3), downsampled resolution (128^3), with a TF designed to reveal bones, heart, and kidneys.
- **Aneurysm dataset:** Original resolution (256^3), downsampled resolution (64^3), with a TF designed to visualize blood vessels.
- **Head dataset:** Original resolution (512^3), downsampled resolution (128^3, with a TF designed to reveal bones and skin.

4.3.2 Analyzed methods

In the following paragraphs, the following methods will be analyzed: subsampling, averaging, Gaussian filtering, bilateral filtering, and topology-guided downsampling [43]. Along with the presentation of each method, a schematic figure is shown in 2D to facilitate its understanding.

Subsampling



Subsampling refers to the method that simply selects a subset of the original samples as the representative voxels for the reduced model. In our case, the representative voxel is one of the voxels in each subvolume of 2^3 voxels. We always select the same voxel of each subvolume, which means that the representative voxels are taken at regular intervals from the voxels of the high-resolution dataset. On the one hand, the main advantage of this method is actually the little amount of data processing required, which only consists in the selection of the representative voxel. On the other hand, the drawbacks of this method are the exaggerated staircase artifact it generates in the downsampled models, and the evident loss of features (see Figure 4.1).

We did experiments with several strategies for the selection of the representative value. For instance, we made some tests by selecting the high-resolution voxel with maximum intensity, minimum intensity, most different intensity, and averaging values using the densities itself as weights. Some of the results obtained from these experiments are shown in Section 4.3.4.

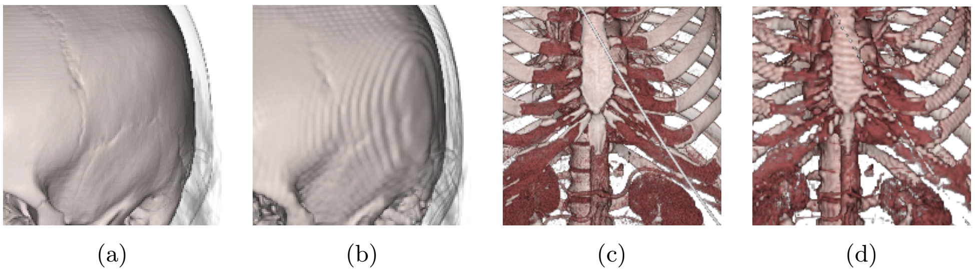
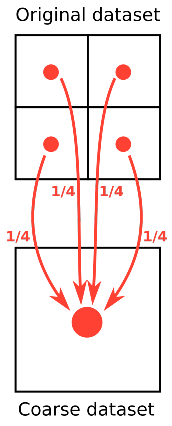


Figure 4.1: The subsampling method generates staircase artifacts (b) with respect to the original dataset (a). Furthermore, thin structures like the catheter disappear in the downsampled model (d) whereas they are visible in the original resolution one (c).

Averaging



The averaging filter simply consists in creating a new value by averaging all the values from the original volume. In our case, for one-level subsampling, this consists in averaging the eight values (2^3 subvolumes) of the upper level. This requires a little more computational effort than simple subsampling, as the representative value in the lower resolution levels is the result of a computation involving eight voxels. As shown in Figure 4.2, the results generated by this downsampling method have more aggressive loss of features than plain subsampling, but provide a smoother look, partially avoiding the staircase artifact seen in subsampling.

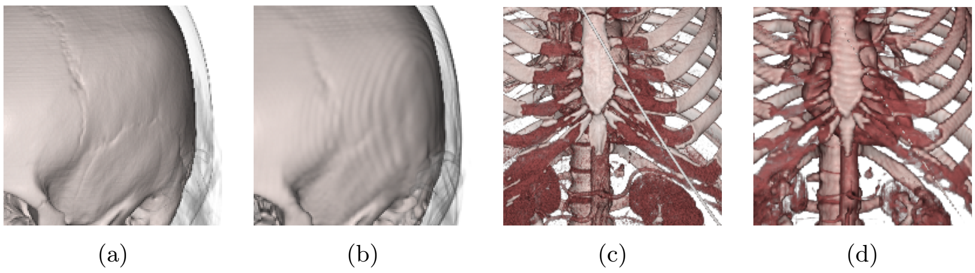
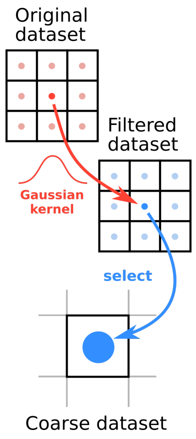


Figure 4.2: Averaging still presents staircase artifacts in the lower resolution representations (b) if compared with the original dataset (a). However, the results are smoother than plain subsampling without previously filtering the dataset to downsample. Furthermore, it does not preserve some features of the original dataset (c) in the downsampled model (d).

Besides those two simple methods, more elaborated techniques include Gaussian filtering and bilateral filtering. Both techniques have been extensively used in image processing algorithms. The core feature of Gaussian filtering is the noise reduction, at the cost of blurring the edges. On the contrary, the bilateral filter is intended to preserve edges. In the following, we introduce both filters using the notation by Paris et al. [69].

Gaussian filtering



Gaussian filtering is a typical operation in image processing that consists in applying a low pass filter over the image samples. Given an image I, its filtered version is computed as a weighted average of the neighboring pixels. So pixel p will have a value that depends on the pixels around a neighborhood q of the original pixel. Typically, this is denoted as:

$$GF[I]_p = \sum_{q \in S} G_\sigma(|p-q|) I_q,$$

where S is the kernel support and G is a 2D filter kernel:

$$G_\sigma(x) = \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2}{2\sigma^2})$$

To downsample volumetric models, we apply this transformation in 3D, so the neighborhood, instead of consisting of a rectangular region, corresponds to a cube. This process incurs a higher data processing than the previous methods. Its cost depends on the radius of the kernel, which determines the span the filter takes in the spatial domain. The representative values in low-resolution voxels are the filtered high-resolution voxels taken at regular positions. Summarizing, the original resolution dataset is first filtered and then subsampled to obtain the low-resolution dataset. The results obtained by the Gaussian filter are a little smoother than previous methods. As shown in Figure 4.3, it slightly alleviates the staircase artifacts with respect to averaging. However, it does not preserve some features, which are clearly visible in the original model.

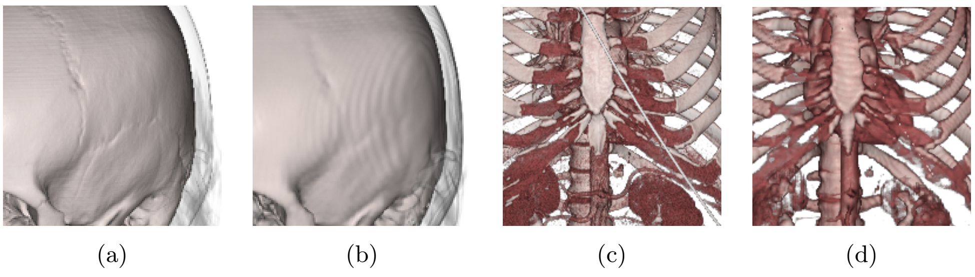
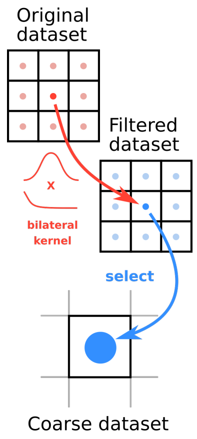


Figure 4.3: Gaussian filtering provides smooth results (b), similar to the original dataset (a). However, like the previous methods (e.g., averaging) it fails at preserving some features in the downsampled dataset (d) that are visible in the original one (c).

Bilateral filter



Similar to the Gaussian filter, the bilateral filter is a weighted average of a neighborhood of pixels. The main difference is that the bilateral filter, besides using a spatial Gaussian kernel, takes into account the variation in the values of the pixels to preserve the edges. The rationale behind this is that two pixels should be weighted similarly, not only if their positions are similar, but also if their intensities are comparable too. This results in a formulation such as:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(|p-q|)G_{\sigma_r}(I_p - I_q)I_q$$

where W is a normalization kernel defined as:

$$W_p = \sum_{q \in S} G_{\sigma_s}(|p-q|)G_{\sigma_r}(I_p - I_q)$$

The bilateral filter appeared in several independent publications [1, 78, 81], and has since then further improved, especially for speed [69]. Anyhow, the amount of data processing required for this method is higher than for the Gaussian filter, in which optimizations are easier to implement. The steps to follow in

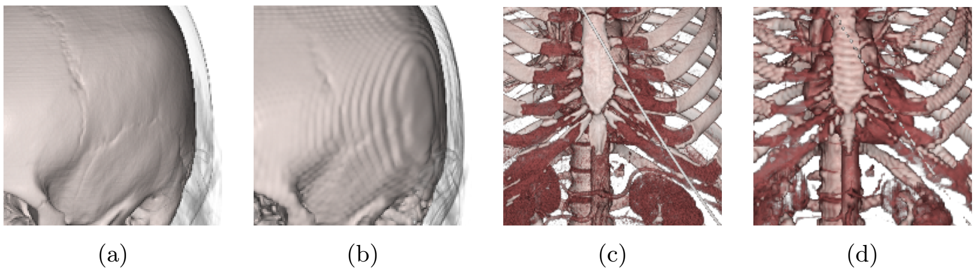


Figure 4.4: The bilateral filter achieves results very similar to plain subsampling. It is a noise reduction filter that does not avoid aliasing artifacts. As shown in (b), the low resolution dataset still exhibits staircase artifacts that are not present in the original dataset (a). Furthermore, thin structures like veins (c) are not preserved convincingly after downsampling, in the low resolution dataset (d).

the implementation of this downsampling method are analogous to those in the Gaussian filter: the original data first needs to be filtered and then subsampled. The representative values in lower resolution voxels are values taken from the original dataset after having been filtered. Not surprisingly, this filter has no major effect in downsampling, as it is aimed at noise reduction when its parameters are properly adjusted, and does not reduce the aliasing problems that lead to the staircase artifacts seen in plain subsampling (see Figure 4.4). As a consequence, the results it provides with the tested models are quite similar to the ones achieved by plain subsampling.

Topology-guided downsampling

Topology-guided downsampling is a method by Kraus and Ertl [43] that tries to preserve the topology of the scalar field as much as possible by preferably selecting the values of critical points. To do this, every voxel in the original dataset is classified as a regular, saddle, or extremum point. After this classification, the most representative sample in the neighborhood of each voxel is taken during downsampling. Although their technique is optimized to perform efficiently, the amount of data processing required is high compared to the previous filters. The results achieved by this technique are quite good in terms of the preservation of features. The original topology is preserved better than classic filters, as seen in Figure 4.5. However, the results provided by this method are bumpy and distorted, and usually enlarges (mainly because of the loss of resolution) or displaces structures more than desirable.

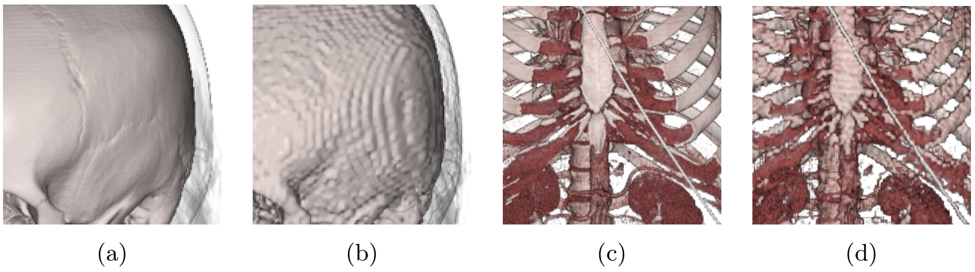


Figure 4.5: Topology-guided downsampling is a feature-preserving downsampling method. Thin structures like veins in (c) are preserved after downsampling the model (d). However, these structures are exaggerated or displaced due to the selection strategy of the representative voxel. Moreover, it can be seen that the smoothness of the original surfaces (a) is not preserved in the downsampled representations (b).

4.3.3 Results and quality assessment

Note that, among the described methods for the downsampling of medical models, some choices that work properly for other purposes in 2D images such as the bilateral filter, and others that aim at the preservation of features such as the topology-guided downsampling, produce worse results than simpler approaches such as the Gaussian filter or the average filtering. Even just subsampling the model seems to produce comparable or even better results, in some cases, than those approaches. This is in part due to the fact that the relation between the filtered volume and the final rendered image is TF-dependent and highly non-linear, with the consequence that such successful image processing techniques may exhibit poor performance in terms of the quality of the final images. Chapter 6 performs an analysis of the relation between the TF and the variation of densities of the scalar field among the original resolution dataset and the coarser resolution datasets. The visualization artifacts due to the loss

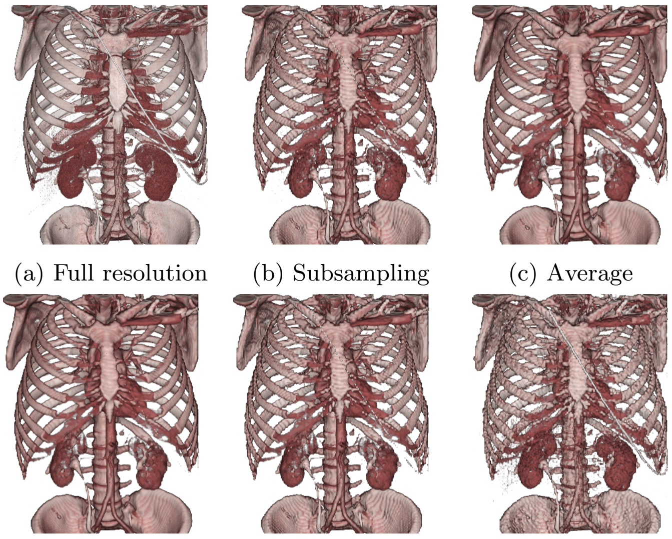
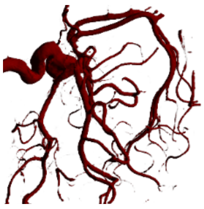
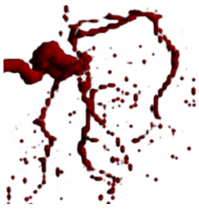
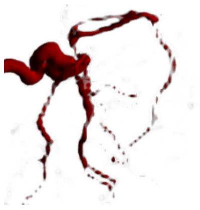
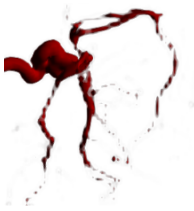

(a) Full resolution (b) Subsampling (c) Average
(d) Gaussian (e) Bilateral (f) Topology-guided

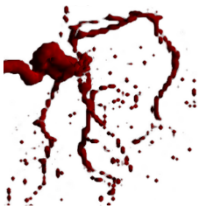
Figure 4.6: Comparison of downsampling filters for volumetric scalar fields. (a) shows the full resolution (512^3) CT dataset. The other models (128^3) are produced using different filters. Plain subsampling (b) loses small features and does not produce smooth results. Applying average (c) and Gaussian (d) filters before subsampling achieves smoother results, but still fine structures are lost. A bilateral filter (e) is basically achieving results almost identical to (b). Topology-guided downsampling (f) better preserves the original topology but the results are undesirably rough and not practical for medical purposes.


(a) Full resolution


(b) Subsampling


(c) Average


(d) Gaussian


(e) Bilateral

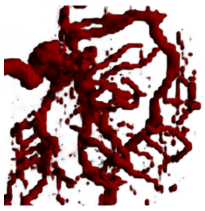
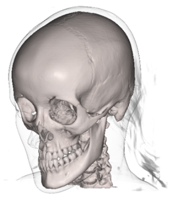
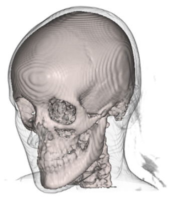
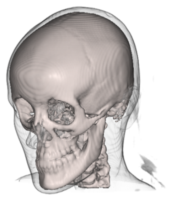
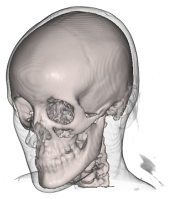

(f) Topology-guided

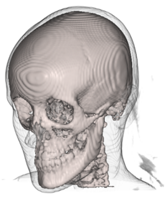
Figure 4.7: Performance of several downsampling filters for the aneurysm model (256^3), which has a lot of small details. The downsampled datasets have a resolution of 64^3 voxels. Most downsampling methods lose a lot of details (b, c, d, and e) or result in overly exaggerated features (f), leading to undesirable visual artifacts.


(a) Full resolution


(b) Subsampling


(c) Average


(d) Gaussian


(e) Bilateral

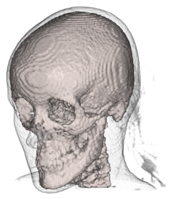

(f) Topology-guided

Figure 4.8: Effect of the tested downsampling methods on the head dataset (512^3). The downsampled datasets have a resolution of 128^3. The algorithms applied are (b) simple subsampling (i.e. taking samples from the original dataset), (c) averaging the 8 voxels from the higher resolution model, (d) Gaussian filtering ($\sigma = 0.7$ in voxel units), (e) bilateral filter (here we take $\sigma = 0.7$ in voxel units, and $\sigma = 6$ for the intensity-based Gaussian, in a range from 0 to 255, and (f) topology-guided downsampling. Finally, (g) corresponds to our feature preserving downsampling method. The examples show that none of the most elaborated previous filters improves over the average or Gaussian filters. In addition, topology-guided downsampling generates really bumpy results.

of resolution are more visible on models that have small features such as the body model in Figure 4.6, or the aneurysm model in Figure 4.7, where the loss of small features is evident. They can also be observed in models with smooth surfaces that exhibit an aliasing problem in their downsampled versions, showing a staircase artifact like in Figure 4.8. Furthermore, even though Topology-guided downsampling preserves some fine details, it produces undesirable, bumpy images, because its sampling strategy modifies the location of the critical points. Note also that even when features are preserved, they tend to become thicker or larger, due to the loss of resolution.

4.3.4 Density-aware selection of representative values

After testing and analyzing the previously mentioned methods, we implemented some new tests before going in the right direction to achieve a feature preserving filter. Most voxel models we deal with, contain density values of the human tissue. A proper feature preserving filter should select the most representative density value for the coarse resolution voxels, so that its visualization resembles the original dataset as much as possible.

Under the assumption that denser tissues are usually more relevant, the first test we implemented consisted in a simple selection rule: taking the densest voxel values from the original dataset. The idea is quite similar to plain subsampling, but instead of taking samples at regular intervals, the representative coarse voxel values are selected from the neighborhood of the original dataset using a simple criterion (maximum density). Figure 4.9-c shows a rendering of a coarse model obtained by using this filter. The final renderings seem to preserve fine structures that disappeared using several filters analyzed previously (e.g., averaging, see Figure 4.9). However, the selection criterion is biased towards denser values, provoking an undesirable thickening effect on dense structures. Furthermore, the filter is not fair with less dense, thin structures, that other TFs might want to reveal. In this case, those structures would probably disappear, as denser ones would have been preserved in its place.

Another approach we tested was inspired by the good behavior on preserving some features of the previous approach, but in the spirit of the averaging filter, in order to achieve smoother results. This approach performs a weighted average of the neighboring voxel values in the original model, using the very densities as weights, so denser values contribute more to the final density value. As we can observe in Figure 4.9-d, details like the catheter and the ureters are

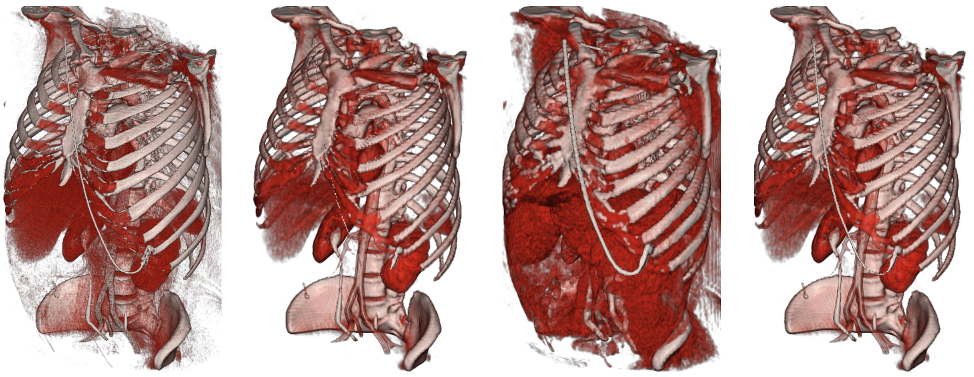

(a) V_0 (b) V_2 average (c) V_2 max. density (d) V_2 weighted avg.

Figure 4.9: Experiments with downsampling filters considering densities. Image (a) shows a rendering of the original dataset V_0. Image (b) shows the rendering of a coarser model V_2 downsampled with a standard averaging filter. In image (c) the downsampled model V_2 was computed using a filter that selects the voxel of maximum density in the neighbourhood. Finally, for image (d) the used downsampling method was an averaging filter benefiting those voxels in the neighbourhood with a higher density. Note that, in this case, results shown in (d) are better than the previous alternatives, conserving features not present in (b), but not thickening thin structures and introducing extra noise as in (c).

preserved such as in Figure 4.9-c, but the structures are not thickened, and the results look quite similar to the rendering of the original dataset (Figure 4.9-a). This method, however, is still biased, so dense values have more importance over less dense ones. Obviously, renderings generated with TFs designed to visualize less dense structures (e.g., air in small lung cavities) will probably not exhibit the same good behavior as in this example.

4.4 Feature-preserving downsampling filter

To overcome the problems concerning the previously presented methods, we have developed a new, Gaussian-based downsampling method that attempts to preserve small features and still produces smooth results at a low cost, and that requires no parameterization, so it runs completely unattended, without any need for user intervention.

In order to preserve details, our algorithm first simulates a downsampling step based on Gaussian filtering and then uses this information as a guidance image in the next step. Then, the real downsampling is performed, using the guidance image previously computed to give more importance to those regions that would previously suffer from excessive degradation so that features are preserved. Another important factor is that the process should run unattended, that is, it should not require manual tweaking of parameters for good results. The rationale behind this is that the filter is intended for use with medical models, to rapidly downsample larger models that would not fit into the GPU. This downsampling should be done automatically, since no expert supervision can be carried out, and fast, so that the physician has the data ready as soon as possible. Moreover, the process must be robust to different models.

Summarizing, the main goals of our downsampling filter are: the preservation of details, achieving an automatic execution, and a low-cost performance.

Given a volumetric scalar field V_0, to compute a coarser representation V_k ($k > 0$) our downsampling technique proceeds in three steps:

- First, a temporary coarser volume S_k is computed by means of Gaussian-filtering and subsampling.
- Next, the difference between V_0 and S_k provides hints about the loss of features in the first step. Using this information we generate a filtered volume F_0 using local kernels that better preserve original volume details that would otherwise disappear with standard Gaussian filtering and subsampling.
- Finally, F_0 is subsampled to obtain V_k.

To compute the filtered volume dataset F_0, we perform the following convolution:

$$F_0(x) = \sum_{i\ \in B_r} V_0(x+i) \cdot f_x(i)$$

where f_x is what we call *Local Feature Kernel.* It is a normalized, feature-preserving kernel with support B_r, a ball of radius r centered at the origin. f_x is in turn a product of a normalized global Gaussian kernel g and a difference kernel d_x:

$$f_x(i) = \frac{1}{\alpha} \cdot g(i) \cdot d_x(i), \quad \forall i \in B_r$$

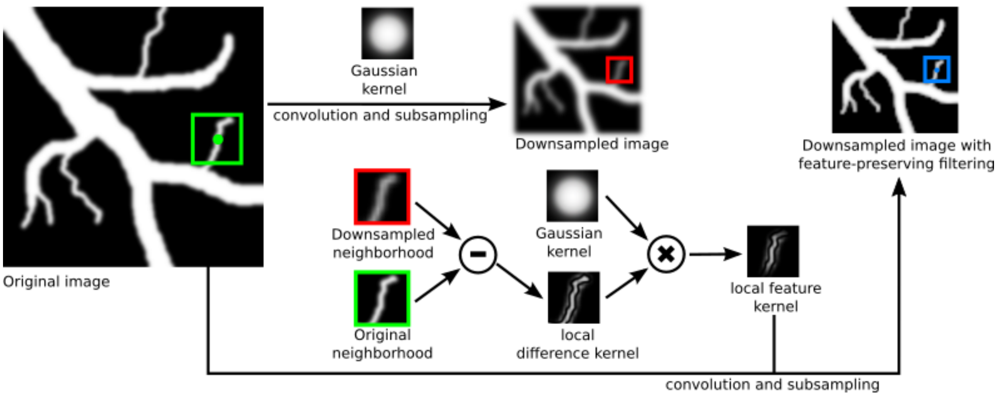


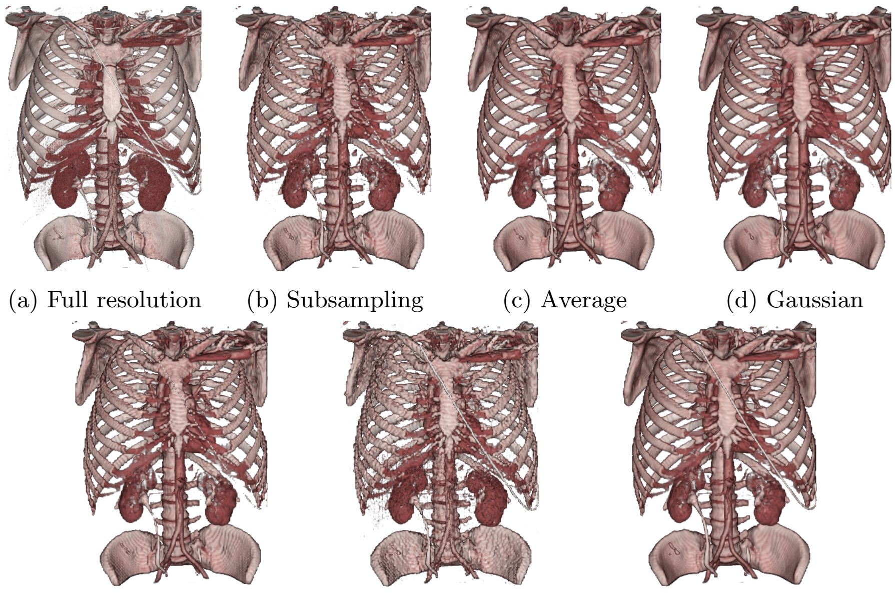
Figure 4.10: Schematic overview of our feature-preserving downsampling filter using Local Feature Kernels. As seen in the image, the technique first needs to perform a Gaussian-based downsampling to generate a guidance downsampled dataset. Then, the original dataset can be filtered with Local Feature Kernels, which are made of a combination of a Gaussian kernel and a difference kernel computed from the sample-by-sample neighborhood differences between the original and the guidance datasets.

The denominator $\alpha = \sum_{j \in B_r} g(j) \cdot d_x(j)$ ensures the sum of weights in f_x equals one. The difference kernel d_x is defined as the normalized absolute difference of values in the neighborhood of x between the original scalar field V_0 and the temporary Gaussian-downsampled scalar field S_k:

$$d_x(i) = \frac{1}{\beta} |V_0(x+i) - S_k(x+i)|, \quad \forall i \in B_r$$

Again, the denominator β ensures the normalization of weights in d_x. Note that V_0 and S_k have different resolutions. Sample positions in the kernel domain happen to be aligned with the center of V_0's voxels, but density values from S_k must be computed by tri-linear interpolation.

Figure 4.10 shows a schematic overview of the algorithm. The difference kernel assigns larger weights to those samples in V_0 that are prone to disappear (those that most differ with S_k). As both g and d_x are combined into f_x, smoothing or sharpening is done depending on their weights, which are local to the filtered sample position x. Homogeneous regions will provoke homogeneous difference kernels, thus giving g_x greater influence, whereas feature regions will provide more characteristic difference kernels for the feature selection task.



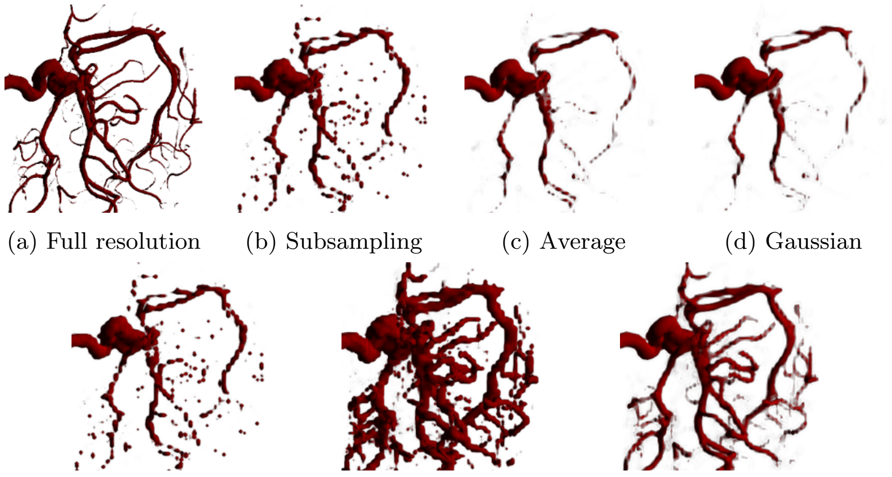
(a) Full resolution (b) Subsampling (c) Average (d) Gaussian

(e) Bilateral (f) Topology-guided **(g) Feature-preserving**

Figure 4.11: Comparison of downsampling filters for volumetric scalar fields. (a) shows the full resolution (512^3) CT dataset. The other models (128^3) are produced using different filters. Plain subsampling (b) loses small features and does not produce smooth results. Applying average (c) and Gaussian (d) filters before subsampling achieves smoother results, but still fine structures are lost. A bilateral filter (e) is basically achieving results almost identical to (b). Topology-guided downsampling (f) better preserves the original topology but the results are undesirably rough and not practical for medical purposes. Our solution (g) is able to preserve details and also maintains smooth surfaces.

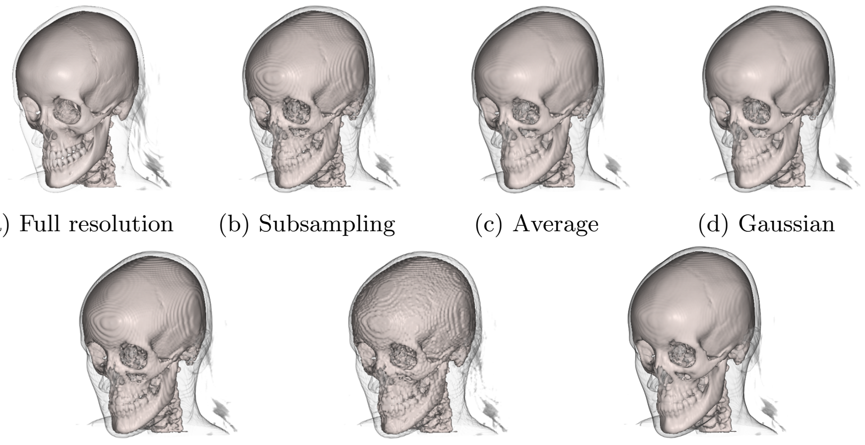
4.5 Evaluation and results

Figures 4.11, 4.12, and 4.13, show examples of the results obtained by the analyzed downsampling filters and by the proposed feature-preserving downsampling filter. While the plain subsampling and Bilateral filters provide noisy results, averaging and Gaussian-filters obtain a much softer look, closer to the original dataset render. All of them, however, fail at preserving thin structures and other details (see the ureter and the catheter in Figure 4.11, and the thin veins in Figure 4.12). Not like these approaches, the topology-guided downsampling method better preserves much of the topology of the original dataset. However, it displaces the original location of critical data samples, thus achieving a nondesirable bumpy look.


(a) Full resolution (b) Subsampling (c) Average (d) Gaussian

(e) Bilateral (f) Topology-guided **(g) Feature-preserving**

Figure 4.12: Performance of our downsampling filter for the aneurysm morel (256^3), which has a lot of small details. The downsampled datasets have a resolution of 64^3 voxels. Most downsampling methods lose a lot of details (b, c, d, and e) or result in overly exaggerated features (f), leading to undesirable visual artifacts. Our method (g) performs better than the others, preserving fine details that are visible in the full resolution model and still providing smooth results.


(a) Full resolution (b) Subsampling (c) Average (d) Gaussian

(e) Bilateral (f) Topology-guided **(g) Feature-preserving**

Figure 4.13: Effect of the tested downsampling methods on the head dataset (512^3). The downsampled datasets have a resolution of 128^3. The algorithms applied are (b) simple subsampling (i.e. taking samples from the original dataset), (c) averaging the 8 voxels from the higher resolution model, (d) Gaussian filtering ($\sigma = 0.7$ in voxel units), (e) bilateral filter (here we take $\sigma = 0.7$ in voxel units, and $\sigma = 6$ for the intensity-based Gaussian, in a range from 0 to 255, and (f) topology-guided downsampling. Finally, (g) corresponds to our feature preserving downsampling method. The examples show that none of the most elaborated previous filters improves over the average or Gaussian filters. In addition, topology-guided downsampling generates really bumpy results. Our method, in this case, behaves similar to the Gaussian filter, providing smooth results, and slightly preserves small pieces of bone under the eyebrows and on the base of the skull.

The feature-preserving downsampling filter just presented is able to preserve those important features that all the previously mentioned methods have not been able to capture in the downsampling process and also conserves the smoothness of the original surfaces. Furthermore, it does not require a precomputation time much higher than affordable. For instance, without any optimization, running on a commodity PC (Intel Core i7 CPU, 8GB RAM) our algorithm takes up to 4 minutes in order to compute the downsampling for the Thorax dataset shown in Figure 4.11 from 512^3 to 128^3 voxels. Although it is a few times more costly than a standard Gaussian-downsampling, it is reasonable for a pre-process and scales linearly with the resolution of the input dataset.

4.6 Conclusions and future work

Since medical models are now commonly available (one can get the resulting data from CT or MRI in CDs, DVDs, or another storage device, after a medical test), the need of rendering such models in commodity PCs has grown. Reducing the resolution of the scalar field data through downsampling techniques is a key step in several scenarios, such as when the data must fit a small GPU, or when the models are huge enough that even high-end GPUs in the market cannot hold it. In these cases, building a multiresolution representation involves downsampling the original datasets to build the intermediate levels. However, the main shortcoming of downsampling is that many fine details and important features are lost in the coarse datasets. With regards to this issue, this chapter has presented the following contributions:

- A description and an analysis of several techniques for the downsampling of volumetric models.
- A new filtering method aimed at the feature-preserving downsampling of volumetric scalar data.

On the one hand, we have identified some flaws in the analyzed filters in Section 4.3. We have seen that either they fail at the preservation of features during downsampling, or they preserve features at the expense of introducing a spatial bias of the preserved features and exhibiting a bumpy look. On the other hand, in Section 4.4, we have presented a novel feature-preserving filter for the downsampling of volumetric scalar data with the intention of solving the previously mentioned issues. Our filter adaptively smooths or preserves features depending on the topology of the surrounding region of the voxel being filtered. We do this by generating Local Feature Kernels. These are the product of a smoothing Gaussian kernel and a local distance kernel derived from the voxel-wise absolute distance between the full resolution dataset and a previous downsampling simulation. Our proposed heuristic identifies zones which are potentially likely to disappear and gives them more importance when filtering. As a result, they are preserved after subsampling.

We have shown that the presented filter improves the quality of several models with respect to other downsampling techniques, and it is especially suited for models with small features, such as the aneurysm model (Figure 4.12). In our experiments, we also found that the improvement over some models was more limited (see the Head model 4.13), but in any case, the quality was at least analogous to Gaussian filtering. Thus, our filter can be safely used in a vast majority of models. Besides the quality of the obtained results, the method is completely automatic and runs unattended, without requiring any intervention from the user. Furthermore, the computation times are low, even without having optimized the algorithm. Another strength is the fact that it is orthogonal to some compression techniques such as wavelets, which could also be applied in order to reduce the memory bandwidth requirements.

Since the key of this filtering method resides in suitably adjusted Local Feature Kernels, we believe that the filter can still be improved. Following this line, we plan to further explore additional ways to determine their weights, variance, and radius. Another possible improvement for the presented technique is its optimization. On this line, some worthy options include the design of a parallel scheme to take advantage of hardware and some research on the separability of the filter.

4.7 Publications

This chapter has originated the following publications:

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez, Frederic Pérez. (May 2015) *Feature-Preserving Downsampling for Medical Images. In EuroVis 2015: The EG/VGTC Conference on Visualization: Posters track. European Association for Computer Graphics (Eurographics)* [15]

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (July 2017) *Downsampling methods for medical datasets. In Proceedings of the CGVCVIP 2017: International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing: Lisbon, Portugal, (pp. 12-20). IADIS Press.* [17]

5 Downsampling and Efficient Storage of Gradient Data

Chapter 4 dealt with the data processing stage of the visualization pipeline, focusing on the downsampling process of the scalar field. Besides the scalar field, to afterward render volume data, gradient directions are needed for shading purposes. If gradients are computed on the fly in GPU shaders by using surrounding densities in the coarse volume, significant visual errors and artifacts can be introduced. Approaching this issue from the data processing stage, to avoid these problems, this chapter handles the shading of coarse datasets through the use of pre-computed gradients. In the following sections, we will present a downsampling filter for pre-computed gradient data and a high-precision, interpolation-friendly, encoding, and decoding scheme for its storage in GPU 3D textures.

5.1 Motivation

One of the main aspects that directly affect the quality of volume visualization is shading. In modern ray casting algorithms, shading can be performed after computing the surface orientation at each sample position by means of the scalar field gradient, which is typically computed on the fly in shader code by evaluating the surrounding densities. In multiresolution visualizations, shading of coarse representations by means of this technique implies computing gradients from a scalar field that differs from the original dataset, a fact that can

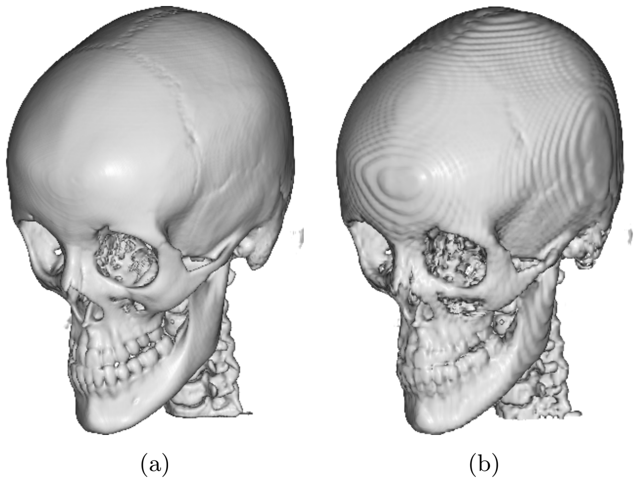


Figure 5.1: Ray casting images of the Head model visualized with a TF designed to show bone surfaces. The left image (a) corresponds to the original dataset (512^3) whereas the image at the right (b) corresponds to a coarser dataset (128^3). In (b), staircase artifacts are visible due to the shading performed using gradients computed on the fly (with central differences) from the downsampled scalar field.

lead to inconsistent shading among different levels of resolution. This fact is amplified because of the well-known error increasing effect when approximating derivatives by finite differences on downsampled representations. Figure 5.1 shows the difference between shading a dataset visualized at its original resolution with respect to a low-resolution representation, computing gradients on the fly in both cases.

5.2 Problem addressed

The problem faced in this chapter is the fact that the quality of the shading in multiresolution datasets is greatly affected by the way in which gradients are computed in coarse levels. As seen in Chapter 4, the downsampling process drastically reduces the resolution of the datasets, thus provoking an inevitable information loss and a modification of the topology of the original scalar field. In Figure 5.2, the effect of downsampling is depicted in a 2D space. It is easily noticeable how the topology of the represented surface gets drastically changed as the resolution decreases. The staircase shape exhibited on the surface in Figure 5.2-b also affects the direction of the computed gradients, not matching the gradients computed from the original dataset anymore. In Figure 5.1-b,

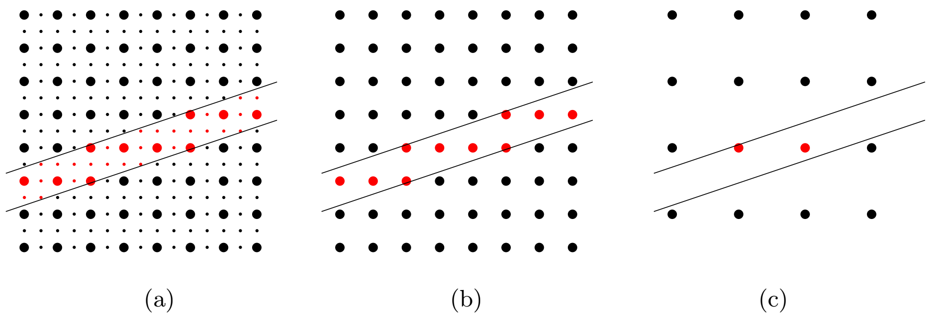


Figure 5.2: Images (a), (b) and (c) depict the effect of iteratively downsampling data (images are shown in 2D for simplicity, although the same concept applies for the 3D case). In (a), a feature/surface is well represented by the boundary between different scalar values. After an iteration of downsampling, image (b) shows how the same feature is not so suitably represented by the current scalar field anymore, which actually yields a more staircase-looking topology. In the last image (c), after another downsampling step, the feature even starts disappearing.

the staircase artifact is mainly visible because the shading is using inaccurate gradients computed on the fly from the coarse dataset.

In order to solve the stated problem, the solution we propose in this chapter is using gradients pre-computed from the original scalar field. We will improve the shading of coarse datasets using these pre-computed gradients after having been downsampled. The proposed improvements in the current chapter are:

- A downsampling filter for pre-computed gradients that better preserves gradient directions.
- A storage technique that improves the precision of downsampled gradients by making better use of the allocated memory.

5.3 Gradient estimators

In order to explore other possibilities to improve the shading of coarse datasets, before approaching the staircase artifacts issue using pre-computed gradients, we performed a few tests estimating the gradients on the fly in our DVR algorithm with some techniques. As explained in Section 3.3, several methods exist to estimate gradients, which differ in the computational complexity and the accuracy of the resulting gradients.

Finite differences

Finite difference schemes are fast to compute, and thus an ideal candidate for real time DVR. Within this family of methods, we performed tests with the two methods, *forward differences* and *central differences*, with similar results in both performance and appearance. Between these two methods, *central differences* has been the preferred method to evaluate gradients on-the-fly during this thesis, as it is the most common approach for gradient estimation in volume graphics. The results it provides are slightly smoother and its computational cost is comparable with the former one, even requiring a different number of texture lookups (probably due to texture cache issues). Each of the three components of the gradient vector $\nabla f(x, y, z)$ is estimated by a central difference, resulting in:

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

which requires six additional samples taken at a distance h around the central sample. For pre-computed gradients, the step size h can directly be the grid size in order to avoid unnecessary interpolation operations. When computing gradients on the fly, the step size h is typically small with respect to the grid size.

The results achieved in shading using *central differences* are shown in Figure 5.1, which exhibits the staircase artifact on the surface of the skull. Smoother results are desirable, which led us trying the well-known gradient estimator explained in the next section.

Sobel operator

The *Sobel* operator is a common discrete filter kernel also used for gradient estimation. The standard 3D Sobel kernel has size 3^3. Thanks to the more intense sampling work, the quality of the estimated gradients is better than using *central differences*. A comparison between the results obtained using *central differences* and the *Sobel* operator is shown in Figure 5.3. Note that the staircase artifacts are now less visible, although not completely avoided. Nonetheless, an obvious disadvantage of a full 3^3 filter kernel such as this is

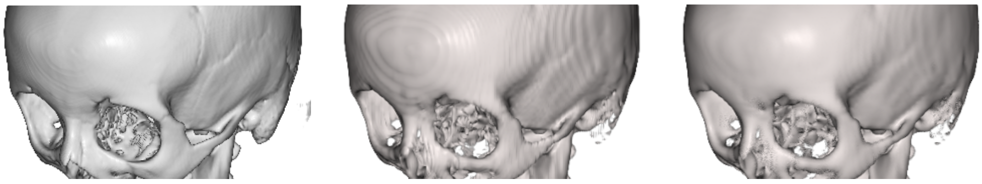

(a) V_0 - Central differences (b) V_2 - Central differences (c) V_2 - Sobel

Figure 5.3: Rendering of coarse datasets estimating gradients on the fly. An image of the original dataset estimating the gradient with the *central differences* method is shown in (a). The other images show a coarser resolution level, shaded with gradients estimated with *central differences* (b) and the *Sobel* operator. Note that the staircase artifacts are less visible in (c) than in (b), although not completely avoided.

its computational complexity. Although an optimized implementation of the filter can be executed in real time in current graphics units, computational power and battery life are critical issues to take into account when dealing with hardware such as mobile devices. Therefore, it is preferable exploring additional ways to improve the quality of the shading of coarse datasets.

5.4 Downsampling of gradient data

In order to avoid costly gradient estimators that improve the quality of shading, we propose using gradients pre-computed from the original scalar field to improve the shading of coarse datasets. To keep gradient directions consistent, we pre-compute a dataset of gradients G_0 from the original scalar field V_0, and we iteratively downsample G_0 to generate coarser representations G_k of the gradients vector fields that match the resolution of the coarse models V_k. Thus, the visualization pipeline for multiresolution datasets used in this chapter uses a three-component 3D texture of pre-computed gradients for the visualization of each coarse dataset.

5.4.1 Naive downsampling approach

However, a naive downsampling of pre-computed gradients without having previously applied an appropriate filter achieves unexpected results (see Figure 5.4). This is due to the fact that the topology of the downsampled dataset has been modified with respect to the original. For that reason, in some cases, regions containing boundaries between materials in the coarse resolu-

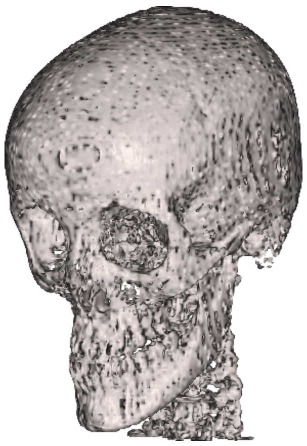


Figure 5.4: This ray casting image is shaded using downsampled pre-computed gradients. The topology of the downsampled scalar field V_k has changed with respect to the original dataset V_0. Therefore, using a naively filtered downsampled dataset G_k of the original pre-computed gradients G_0 that does not take into account any changes in the topology produces these annoying hole-like artifacts.

tion dataset could correspond to regions of a homogeneous material in the high-resolution one. Whenever that happens, the locations of the sampled gradients in the low-resolution dataset correspond to gradients that are not properly defined in the high-resolution dataset, and thus, using gradients that have been downsampled without any further consideration would lead to erroneous visualizations such as in Figure 5.4.

5.4.2 Gaussian derivatives

The previously mentioned issue is basically an aliasing problem between the high-resolution and the low-resolution datasets. To reduce this aliasing problem, we performed some tests with Gaussian derivative operators in order to pre-compute gradients from the original resolution dataset with a wider radius of influence, so that for each voxel of the original dataset, more voxels in the neighborhood are considered to estimate the gradient. Figure 5.5 shows the results of shading a downsampled dataset with gradients pre-computed using several configurations of a Gaussian derivative operator. The results show that the aliasing hole-like artifacts disappear as we use pre-computed gradients that were computed with a larger radius in the operator. For a large enough radius, those annoying artifacts completely disappear. At this point, however, a clear disadvantage is that the level of smoothness is excessive.

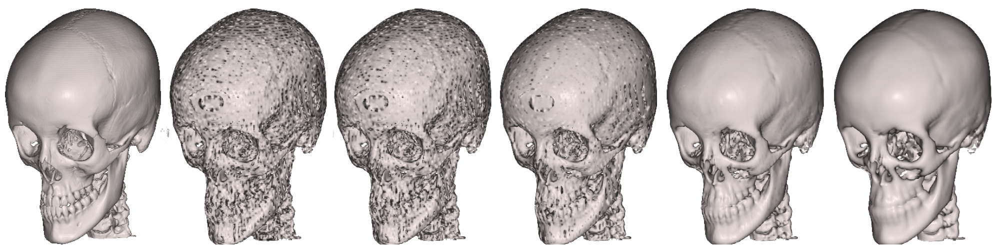


Figure 5.5: Using Gaussian derivatives to shade the coarse dataset (V_2) at 1/64th the original resolution (V_0). Gaussian kernels of different sizes (σ) have been used in order to see their results in the downsampled model. Note that, as we increase the radius of the Gaussian function, the black hole artifacts tend to disappear. However, the results are excessively smooth and lack surface details.

5.4.3 Proposed filter

In the spirit of Gaussian derivatives, our proposal consists in a downsampling filter that takes into account the magnitude of the gradients within the support of a certain kernel. The proposed filter performs a convolution over a certain dataset of pre-computed gradients G_k with the following kernel:

$$G_k^f(x) = \frac{1}{\beta} \sum_{i \in B_r} G_k(x+i) \cdot m(x+i) \cdot g(i) \tag{5.1}$$

where $m(x+i)$ is the magnitude of the gradient at the neighboring position $x+i$, $g(i)$ is a Gaussian function that gives more importance to those samples nearer to center of the kernel support B_r of radius r, and $\beta = \sum_{i \in B_r} m(x+i) \cdot g(i)$ is the sum of all weights to ensure a normalized contribution of the gradients. For the sake of clarity, we have expressed the equation in 1D, although the same definition applies to the 3D case.

Notice the similarity of equation 5.1 with a bilateral filter [81]. Bilateral filters act essentially as a standard domain filter, averaging values that are similar to the value at the kernel center. The main difference between the bilateral filter and ours is that we are not giving importance to the value at the kernel center, assuming that the gradient at that point might be poorly defined, but giving importance to gradients in the kernel support that inform about a well-defined material boundary. After our filter is applied, the filtered output G_k^f can be safely subsampled to obtain the next coarser dataset G_{k+1} in the multiresolution hierarchy.

In order to pre-compute the whole multiresolution pyramid of gradient datasets, the first step consists in pre-computing the gradients from the original dataset, thus obtaining a new volume dataset of gradient data G_0 with the resolution of the original scalar dataset V_0. Note that these pre-computed gradients G_0 can be estimated with any method. In our case, we have used the *central differences* finite method and achieved good results, as we will see in Section 5.6. By applying the proposed filter and then subsampling, the next level of the multiresolution hierarchy G_1 is obtained from G_0. Following the same procedure, G_2 is obtained from G_1, and subsequent levels are obtained iteratively until the whole pyramid is generated. Furthermore, as we generate each coarse level directly from its direct parent, the support of the filter kernel does not need to be too large. We have used kernels of 3^3 with satisfactory results.

Section 5.6 shows some examples of volume images generated using pre-computed gradients that have been extracted from the original dataset and downsampled with the proposed filter. The presented images demonstrate how using quality gradients obtains better results and makes the staircase artifacts mentioned before disappear.

5.5 Gradient data storage

Before being used in the GPU by the visualization algorithm, we pre-compute and downsample gradients in the CPU, using a floating point representation to conserve as much precision as possible. After this pre-processing step is done, gradient data is ready to be transformed into a more suitable representation that will be translated to the GPU memory, from where our visualization algorithm will be able to fetch gradient information to perform shading effectively.

Observe that, in the following, we will deal with gradient directions, which are important to perform shading operations. Besides gradient directions, gradient magnitudes, although used for some purposes like modulating the contribution of shading on the final surface coloring (i.e., large, well-defined gradients, will be more reliable to perform shading calculations than small, weakly defined gradients), will not be considered in our rendering algorithms. In what follows, therefore, we will only be interested in encoding and retrieving gradient directions (not magnitudes) in the Gauss sphere [22].

As said, in order to use these gradients from shaders in the GPU, we need to choose a proper representation to transmit data into the GPU memory. We will make use of 3D textures to store per-voxel information. At this point, on the one hand, it is important to choose an encoding scheme that avoids losing precision and makes an efficient use of the memory. On the other hand, recalling that the visualization algorithm will take samples at arbitrary positions, the selected encoding scheme will need to be suitable for hardware-assisted interpolation.

5.5.1 Spherical coordinates

The first idea we approached in order to represent gradient directions was using a simplification of spherical coordinates. Spherical coordinates, in our case, need only two values: a polar angle and an azimuth angle which denote a direction in 3D. Spherical coordinates typically require an extra value, radius, in order to represent points in the 3D space. We are, however, not interested in representing locations in space, but directions, and thus this last parameter can be omitted. With only two values, spherical coordinates have the benefit of having a smaller memory footprint than other encoding schemes. Another benefit of this scheme is the lack of redundancy it provides, as no repeated directions can be obtained with different combinations of the angles. However, a problem of this representation is that the distribution of gradient representations is not uniform over the space, having many more gradients grouped towards the poles of the sphere. However, the main shortcoming of spherical coordinates, and the reason why it is unusable in the case of our volume rendering algorithms, is the fact that different gradients cannot be interpolated properly in the direction of the azimuth angle. For instance, the interpolation of two similar directions like $(1, 0)$ and $(359, 0$, could result in a direction point to the opposite direction.

5.5.2 XYZ components

A widely accepted strategy is the use of 8-bit component RGB textures (to store the X, Y, and Z components of a vector) since they are typically precise enough for most applications. However, storing gradients this way has two problems: first, the precision is reduced due to the limited number of bits allocated, and second, much space is wasted if we take into account that several 24-bit RGB combinations lead to repeated gradient directions (e.g., vectors $(1, 1, 1)$

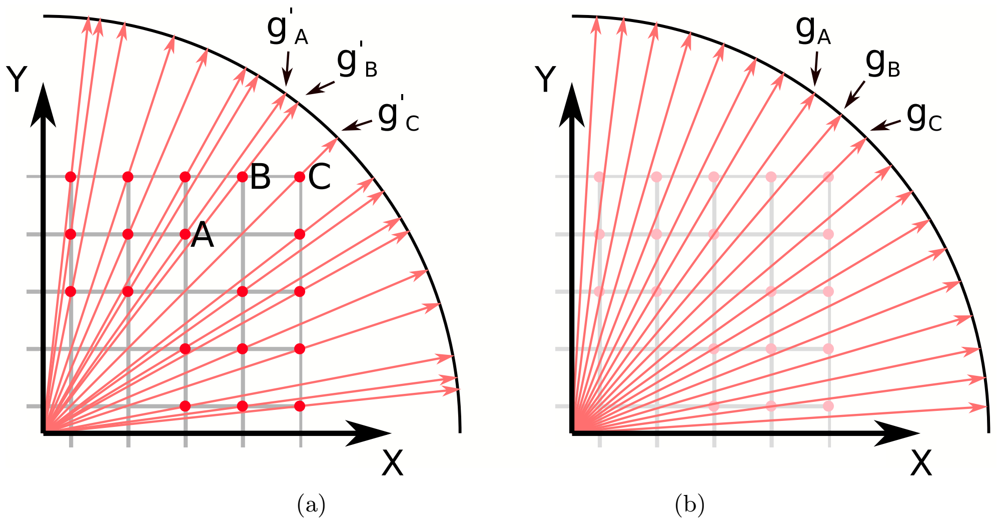


Figure 5.6: Gradients stored into a given texture (drawings are in 2D for clarity) are quantized to fit the bit size of the components. In the case of using 8 bit components, this quantization (a) limits the representable space of gradient directions. Furthermore, the distribution of gradient directions does not fill the representable space uniformly. The method described here applies a transformation on the pre-computed gradients recovered from the texture so that the final distribution of directions becomes more uniformly distributed (b). The result is that quantized values A, B and C encode the uniform directions g_A, g_B i g_C instead of the uneven directions g'_A, g'_B i g'_C that directly correspond to A, B, and C.

and $(2, 2, 2)$ represent the same direction). Furthermore, by storing gradients this way, the distribution of representable directions when queried from the 3D texture is not uniform as shown in Figure 5.6. We could consider using floating point textures, but they require much more space, and they incur in a performance penalty at the time of performing texture fetches, as compared to typical 8-bit RGB textures.

In this section, we propose an encoding scheme that is able to maximize the representable space of gradient directions when storing them into an *RGB* 3D texture of byte precision components. For that purpose, as a pre-process, pre-computed gradients are encoded with a transformation T and quantized before being stored into the GPU *RGB* 3D texture. Then, the visualization algorithm is able to perform texture lookups to recover the encoded gradients, and perform a fast decoding transformation T^{-1} in the shader code to obtain the final gradients that will be used for shading, which better match the original ones. Figure 5.7 shows a graphical overview of the proposed encoding/decoding approach.

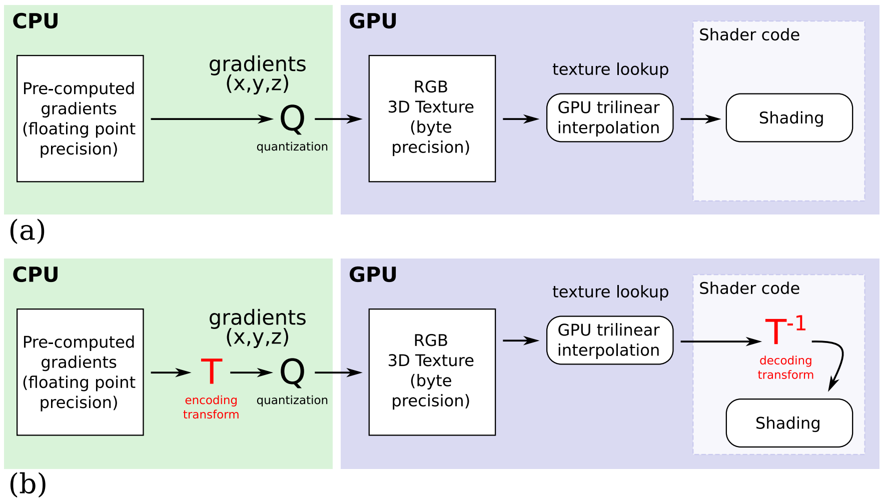

CPU
Pre-computed gradients (floating point precision)
gradients (x,y,z)
Q
quantization
GPU
RGB 3D Texture (byte precision)
texture lookup
GPU trilinear interpolation
Shader code
Shading
(a)
CPU
Pre-computed gradients (floating point precision)
gradients (x,y,z)
T
encoding transform
Q
quantization
GPU
RGB 3D Texture (byte precision)
texture lookup
GPU trilinear interpolation
Shader code
T-1
decoding transform
Shading
(b)

Figure 5.7: Typically, visualization methods that use pre-computed gradients apply a direct quantization of the normalized floating point vectors (a) in order to store them into a byte-precision *RGB* 3D texture. This is an important step where gradients lose precision. We propose a transformation T that is able to maximize the representable space of gradients obtained from a byte-precision *RGB* 3D texture. We encode the original gradients and quantize them in a smart way to optimize the usage of the encoded space. Decoding is performed in the GPU using the inverse transformation T^{-1}.

5.5.3 Monotonic gradient encoding

In order to represent a vector in 3D space, the most common approach is to use three values to specify its X, Y and Z coordinates. The three-dimensional vector space described by these values has the shape of a cube if represented as a point cloud, as seen in Figure 5.8-a. Once these vectors are given a common origin and normalized, the set of points that represent gradient directions can be shown in the Gauss sphere, Figure 5.8-b. In what follows, gradient points projected onto the Gauss sphere will be named *sphere-dots*. We can notice that there are several patterns of empty regions (that is, regions without *sphere-dots*) onto the surface of the sphere (see Figure 5.8-b). Those empty regions correspond to gradient directions that cannot be directly represented with the tree coordinates X, Y and Z due to their bit size precision.

To alleviate this issue and to optimize the representable space of gradient directions, we propose to perform a transformation of the *sphere-dots* to achieve a more uniform distribution over the surface of the Gauss sphere. The solution we propose reduces the *biggest hole* on the surface of the sphere (that is, the

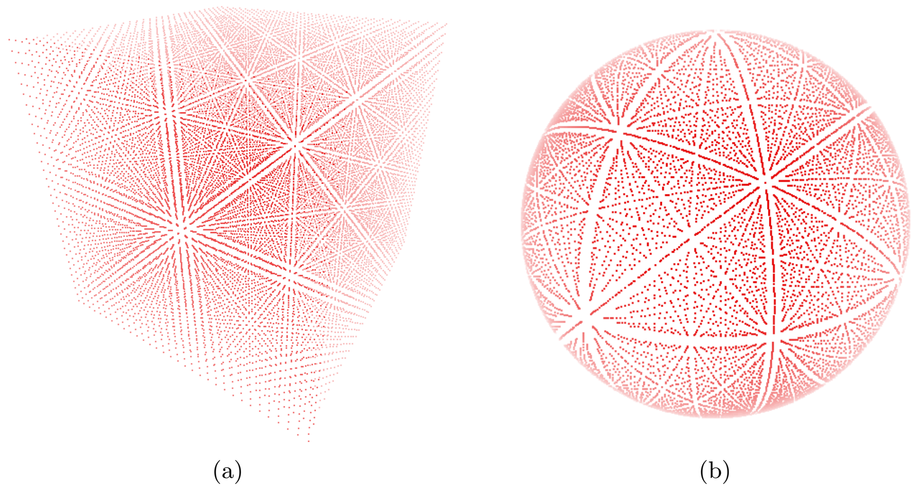


Figure 5.8: Point cloud represented by three values XYZ. (5 bits per value are used in this case to avoid cluttering and better illustrate the empty patterns on the surface). In (a) the points are evenly distributed in the 3D space. On the other hand, (b) shows the corresponding *sphere-dots* after projecting the point cloud onto the surface of a unit sphere.

region with the biggest separation among dots), minimizing the maximum angle achieved between two neighboring gradient directions.

As the pre-computed datasets of gradients are stored into a 3D texture in the GPU, and the shader code of the ray casting algorithm queries this texture to evaluate gradients at any position using hardware-enabled tri-linear interpolation, it is important that the transformation applied to each dot is *monotonic*. With this condition, dots do not get mixed on the surface of the unit sphere (Figure 5.8-b). In other words, the relative position among dots on the surface of the sphere before applying the transformation should not change after applying the transformation. If this condition was not satisfied during this operation, interpolated vectors could be wrong at the moment the transformation takes place in the GPU. The condition of monotonicity (see Figure 5.9) ensures that if we sample the texture using tri-linear interpolation, the decoded results will provide gradient directions that remain within the decoded directions of the surrounding voxel centers.

If we project the point cloud in Figure 5.8-a onto the surface of a cube (instead of a sphere), we obtain the distribution shown in Figure 5.10. The points projected on the surface of the unit cube will be named *cube-dots* from now on to distinguish them from the points on the sphere. With *cube-dots*,

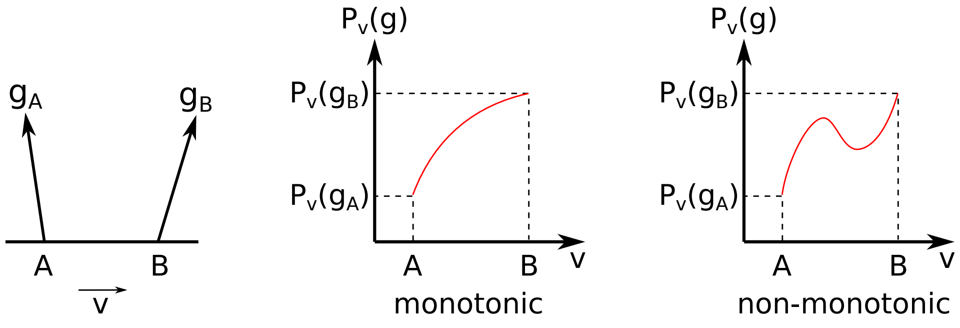


Figure 5.9: Condition of monotonicity. $P_v(g)$ is the projection of g in a given direction v ($P_v(g) = g \cdot v$). To ensure that the transformation of a dot (Figure 5.8) is monotonic, this assertion must be true for all possible tangent directions on the surface of the Gauss sphere.

the patterns of empty regions on the Gauss sphere are easily noticeable. With this projection, we can observe that the overall *cube-dots* follow a pattern of a triangular region that repeats itself eight times on each face of the cube. That adds up to a total number of 48 triangular regions on the entire cube. Figure 5.10-b shows one of these 48 regions. The outer part of each triangle has an empty margin, not filled with *cube-dots*. This means that these gradient directions cannot be properly encoded. In order to improve the distribution of normals, we need a set of dots that do not exhibit those holes and whose distribution becomes more uniform.

Thus, our purpose is to treat each of those triangular regions independently, performing a monotonic transformation on each *cube-dot* within the space of its triangle, thus making the empty bands near the edges shrink.

For the sake of clarity, let us start by discussing the decoding of gradient values obtained from the 3D texture of encoded gradients. To convert the retrieved, uneven gradient directions g'_A onto the corresponding uniform directions g_A (Figure 5.6), we must perform the decoding (inverse) transformation T^{-1} of cube-dots in the space of a triangle. This is a simple and GPU-friendly operation. For that reason, each point in the cloud belonging to a certain triangle (one of the 48 on the cube), has its Euclidean coordinates (XYZ) converted into Barycentric coordinates (UVW). We propose the following equation system using barycentric coordinates to transform the dots:

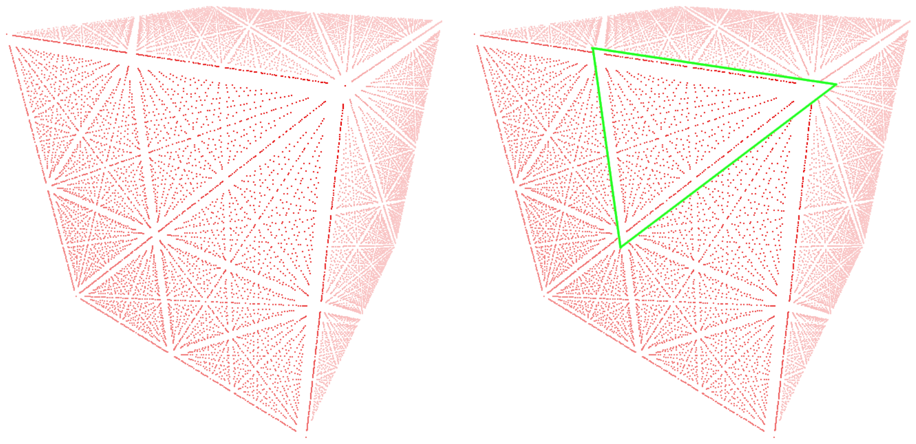


Figure 5.10: In (a), the point cloud represented by three values XYZ projected onto *cube-dots* on the surface of a unit cube (5 bits per value are used in this case to avoid cluttering and better illustrate the empty patterns on the surface). In (b) one of these 48 triangular regions that we have identified on the surface of the cube is highlighted.

$$\begin{aligned}
&\textit{// Attract to vertices}\\
&u1 = \lambda u^2 + (1-\lambda)u\\
&v1 = \lambda v^2 + (1-\lambda)v\\
&w1 = \lambda w^2 + (1-\lambda)w\\
\\
&\textit{// Normalization}\\
&sum = u1 + v1 + w1\\
&u1 = u1/sum\\
&v1 = v1/sum\\
&w1 = w1/sum
\end{aligned} \tag{5.2}$$

where (u_1, v_1, w_1) are the transformed barycentric coordinates. The graphical effect of this transformation T^{-1} on the *cube-dots* is shown in Figure 5.11. Note that, in those equations, different values of λ cause different final distributions of dots. We need to find the value of λ that achieves the best distribution, that is, the optimal value that achieves the best minimization of the empty regions at the boundaries of the triangles. This can be done by measuring the biggest angle between the directions of all pairs of neighboring *sphere-dots* over the whole sphere.

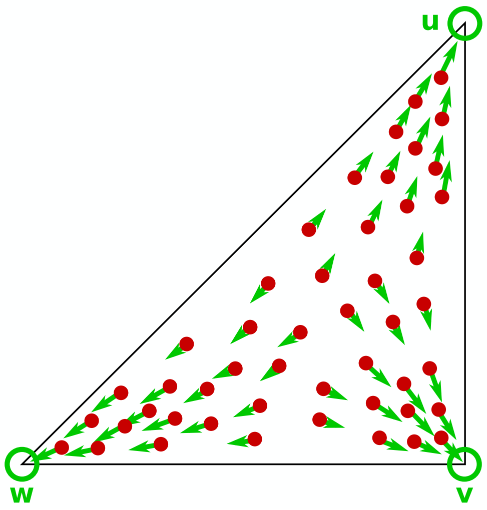


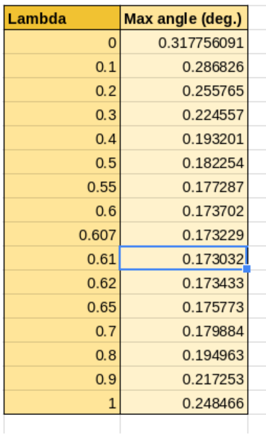
Figure 5.11: The effect of the decoding transformation T^{-1} shown in equation 5.2 can be seen bere. The transformation is applied on the *cube-dots* once they are expressed in barycentric coordinates, and it generates the movement of these dots towards the directions shown by the green arrows, filling the empty areas near the boundaries of the triangle.

Optimization of the Monotonic Gradient Transformation

Figure 5.12 shows that the maximum angle among all possible pairs of neighboring vectors varies as λ goes from 0 to 1. We have found that the optimal value (theoretical minimum) for λ is 0.61, value for which we achieve a maximum angle of 0.1730 *deg*. Table 5.1 shows the maximum angle between two neighboring gradients when not treating the projected dots, using the proposed transformation and the theoretical minimum. The transformation proposed

Method	**Angle**
No treatment	0.3177 deg.
48 regions decoding	0.1730 deg.
Theoretical min.	0.0615 deg.

Table 5.1: Angle denoting the biggest hole in the distribution of dots on the surface of a sphere. These measures have been achieved by generating a triangulation of the *sphere-dots* and taking the diameter of the biggest circumscribed circle among all triangles.



Lambda	Max angle (deg.)
0	0.317756091
0.1	0.286826
0.2	0.255765
0.3	0.224557
0.4	0.193201
0.5	0.182254
0.55	0.177287
0.6	0.173702
0.607	0.173229
0.61	0.173032
0.62	0.173433
0.65	0.175773
0.7	0.179884
0.8	0.194963
0.9	0.217253
1	0.248466

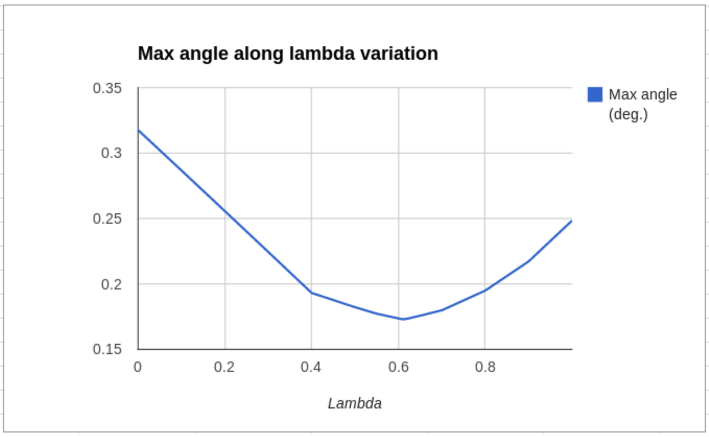


Figure 5.12: Maximum angle (hole) in the sphere of transformed projected points achieved by varying λ from 0 to 1. We can see that the optimal value (theoretical minimum) for λ is 0.61. With this value, equation 5.2 achieves its best distribution of dots, maximizing the representability of gradient directions.

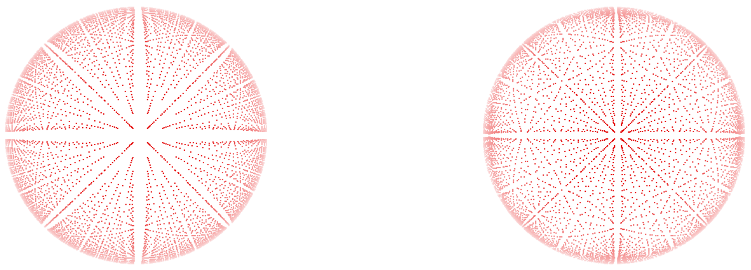


Figure 5.13: Comparison of both cases, untransformed and transformed gradients XYZ (5 bits per value are used in this case to avoid dot cluttering and better illustrate the empty patterns on the surface). Untransformed gradients (a) present empty bands, which represents directions that cannot be encoded. Transforming gradients with the presented method reduces those empty bands, and thus, optimizes the usage of the 3D space to represent more gradient directions.

here achieves a resulting dot distribution that approaches to the theoretical optimum (see the explanation below), which might not be achievable, given the total number of points representable by the three XYZ byte-components.

Figure 5.13 shows the comparison between the *sphere-dots* in both cases: without any treatment and with the proposed decoding transformation T^{-1}. It is clear that empty bands patterns are more evident in the former case.

Computation of the theoretical minimum

Let's suppose that we have 2^{24} points uniformly distributed over the surface of a unit sphere (consider an almost-isotropic distribution of points over the sphere given by an iterative subdivision of a tetrahedron). As the surface of a certain sphere is given by the formula $4\pi R^2$, and $R = 1$ in this case, our sphere will have a surface of 4π. If points are uniformly distributed, as it is the case, a triangulation of this point cloud on the surface will only include triangles that are practically equilateral. In closed triangle meshes, the number of triangles is twice the number of vertices ($T = 2V$). Each triangle surface will be then $4\pi/(2V) = 4\pi/(2 \times 2^{24})$. The maximum angle (in radians) in this case will be the diameter of the circumscribed circle of any of these triangles, and that is because in this distribution of points, we consider all *holes* between points to be equal. We have that the surface S of a triangle circumscribed within a circle of diameter D is $S = \frac{3}{16}\sqrt{3}D^2$. Then:

$$D = 1.756\sqrt{S} = 1.756\sqrt{2\pi}/2^{12} = 4.4/4096\ rad = 0.0615\ deg$$

Thus, if we made a "perfect" use of all the bits in our 8-bit RGB texture, we could store gradient directions with a maximum error of 0.0615 degrees. However, a standard encoding, as shown above, produce errors up to 0.32 degrees, while our encoding approach has a maximum error of 0.17 degrees.

5.5.4 Decoding

The procedure is able to maximize the representable space of gradient directions using three values XYZ of 8 bits. That step actually corresponds to the decoding stage of the pipeline that will take place in the GPU after evaluating the gradient from the 3D texture (see Figure 5.14). The gradient to use in shading operations is, in fact, the one obtained after the transformation T^{-1} takes place. Summarizing, the steps to follow in the decoding stage after obtaining the encoded gradient from the 3D texture are the following:

1. Identify the corresponding triangular region.
2. Obtain the barycentric coordinates of the *cube-dot*.
3. Perform the transformation T^{-1} on the *cube-dot*.
4. Convert the transformed *cube-dot* back to Euclidean coordinates and normalize to obtain the interpolated gradient direction.

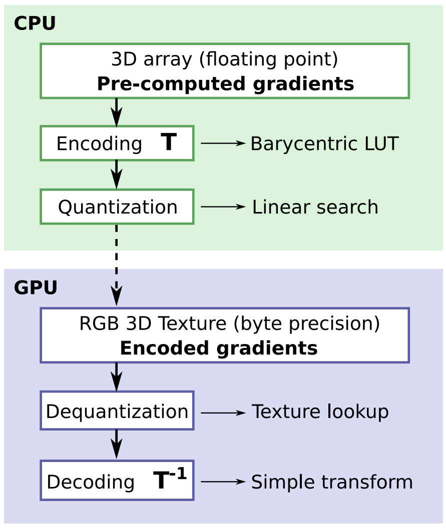


Figure 5.14: Diagram of the tasks in the different stages of the encoding/decoding pipeline. The tasks carried out by the CPU happen in pre-processing time. Encoding gradients require retrieving the transformed values from the pre-calculated LUT, and quantization involves performing a linear search to find the best matching quantized value. In GPU, however, all tasks are fast: dequantization is done at the time of retrieving the encoded value from the 3D texture, and the decoding transformation T^{-1} is a fast calculation involving a few product calculations.

5.5.5 Encoding

Figure 5.14 shows an overview of the gradient encoding/decoding pipeline. The process of encoding gradients T is exactly the inverse to the decoding transformation explained in the previous section. Before performing the encoding process, the gradients we are working with are stored with floating point precision coordinates to avoid a precision loss as much as possible. The steps to follow in order to obtain an encoded vector $\vec{v_e}$ given an original vector $\vec{v_o}$ are quite similar to the decoding operation:

1. Identify the corresponding triangular region.
2. Obtain the barycentric coordinates of the *cube-dot*.
3. Perform the inverse transformation T on the *cube-dot*.
4. Convert the transformed *cube-dot* back to Euclidean coordinates.
5. Quantize the transformed gradient.

Transformation of barycentric coordinates

During the encoding stage, the transformation T of barycentric coordinates in the triangular region must be the inverse of the decoding formulae in equation 5.2 (see the effect of this transformation in Figure 5.11). These equations are a system of three dependent quadratic equations. The easiest way to find the inversion of this system is to proceed with numerical methods. External mathematical packages provide us with tools to solve this problem easily. In our case, we have used *R*, a language designed for statistical analysis. However, this kind of computation is an expensive operation that is better to avoid during the encoding process of many gradients.

Look up table

To reduce the runtime computation cost, we have numerically solved the inversion of this system for a relatively big set of barycentric coordinates in order to construct a look-up table that maps barycentric coordinates to their transformed correspondences (T). Figure 5.15 shows the geometrical representation

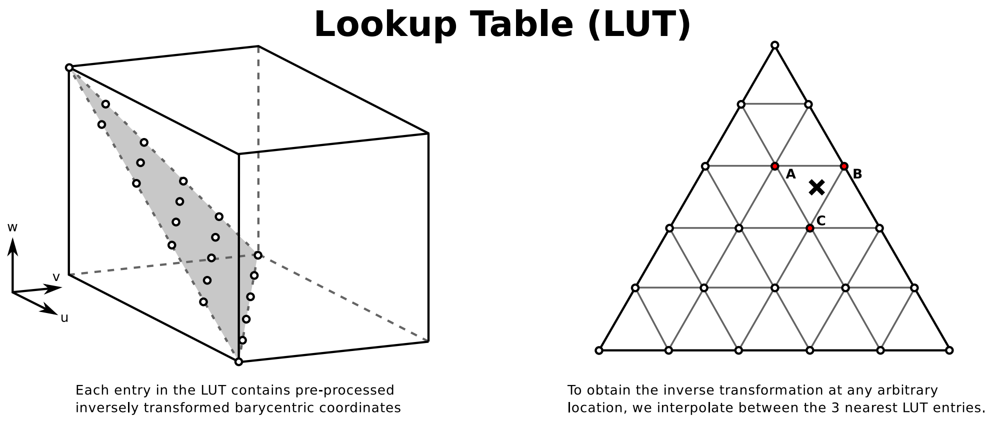

Lookup Table (LUT)
w
v
u
A
B
C
Each entry in the LUT contains pre-processed inversely transformed barycentric coordinates
To obtain the inverse transformation at any arbitrary location, we interpolate between the 3 nearest LUT entries.

Figure 5.15: Barycentric transformation T look-up table (LUT). Each point in the figure corresponds to an entry of the LUT. Entries contain the transformed barycentric coordinates needed for the encoding process. Notice how, despite the fact that barycentric coordinates have three components, only the subspace depicted by the triangular plane contained in the cube is used (barycentric coordinates not belonging in this plane are not normalized and hence they are not useful for us). We can obtain transformed barycentric coordinates at any arbitrary location in the triangle by interpolating the contents of the three nearest entries.

of the look-up table. Points representing valid barycentric coordinates lie on the triangular diagonal plane in the cube. This structure stores transformed barycentric coordinates for several points on that plane. In order to know the inverse barycentric transformation of a certain coordinate, we find its location on that plane and compute the interpolation among the tree pre-computed values stored in the three nearest vertices. By converting the resulting *cube-dot* back to Euclidean coordinates again, we obtain a new high-precision gradient v_l that once decoded is almost equal to the original gradient v_o.

Quantization

In this step, the high precision transformed gradient v_l is downcasted into an 8-bit component quantized gradient v_q. This operation finds the point in the original point cloud (Figure 5.8) that once transformed by the decoding opera-

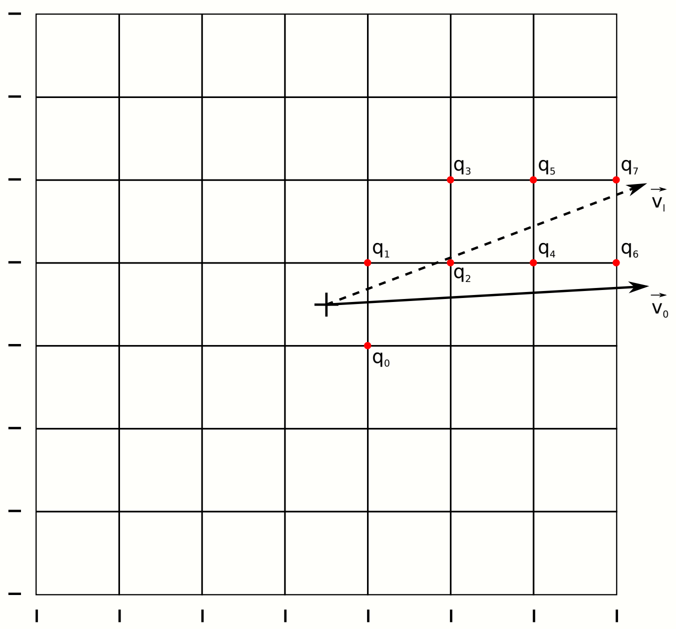


Figure 5.16: Linear search in the quantization process. The transformed gradient v_l is obtained from the original vector v_0 with the help of the LUT. v_l is then used to perform a linear search over a limited subspace of the point cloud. The final quantized point q_i is the one that, after applying the decoding transformation T^{-1}, is more similar to the original vector v_0.

tion, best matches the original floating point precision gradient to be encoded. We use the gradient v_l obtained from the look-up table explained above to perform a linear search over the point cloud, starting at the center of the point cloud and following the direction of v_l. Figure 5.16 shows how the transformed gradient direction (the one achieved by means of the transformation LUT) can be used to greatly limit the search space over the whole possible quantized values, only considering the quantized values surrounding the direction of the transformed gradient. By performing this fast linear search, we select the quantized value which, once decoded, best represents the original gradient direction.

5.6 Evaluation and results

In this section, we have presented a downsampling filter for pre-computed gradients and an encoding/decoding scheme for their storage to maximize the number of representable gradient directions. In the following sections, we are showing the visual results obtained by our downsampling and encoding scheme, its performance (pre-processing time), and some error measures regarding the use of LUTs of different size.

5.6.1 Visual results

Figure 5.17 shows the difference between shading several coarse datasets using gradients computed on-the-fly and using pre-computed, filtered gradients, with respect to the ray casting image of their original datasets. It is easy to see how the downsampled datasets rendered with pre-computed filtered gradients obtain much better results than computing gradients on-the-fly, even when the coarse datasets are the same. This demonstrates the importance of gradients in shading. The staircase artifacts that are visible in the downsampled datasets shaded with gradients evaluated on-the-fly in shader code could be smoothened by increasing the size of the kernel used in the downsampling filter of the scalar field. However, as the size of the kernel grows, the scalar field becomes smoother and smoother, and more features are prone to disappear consequently. Using pre-computed and downsampled gradients makes it possible to remove these undesirable staircase artifacts without sacrificing important features.

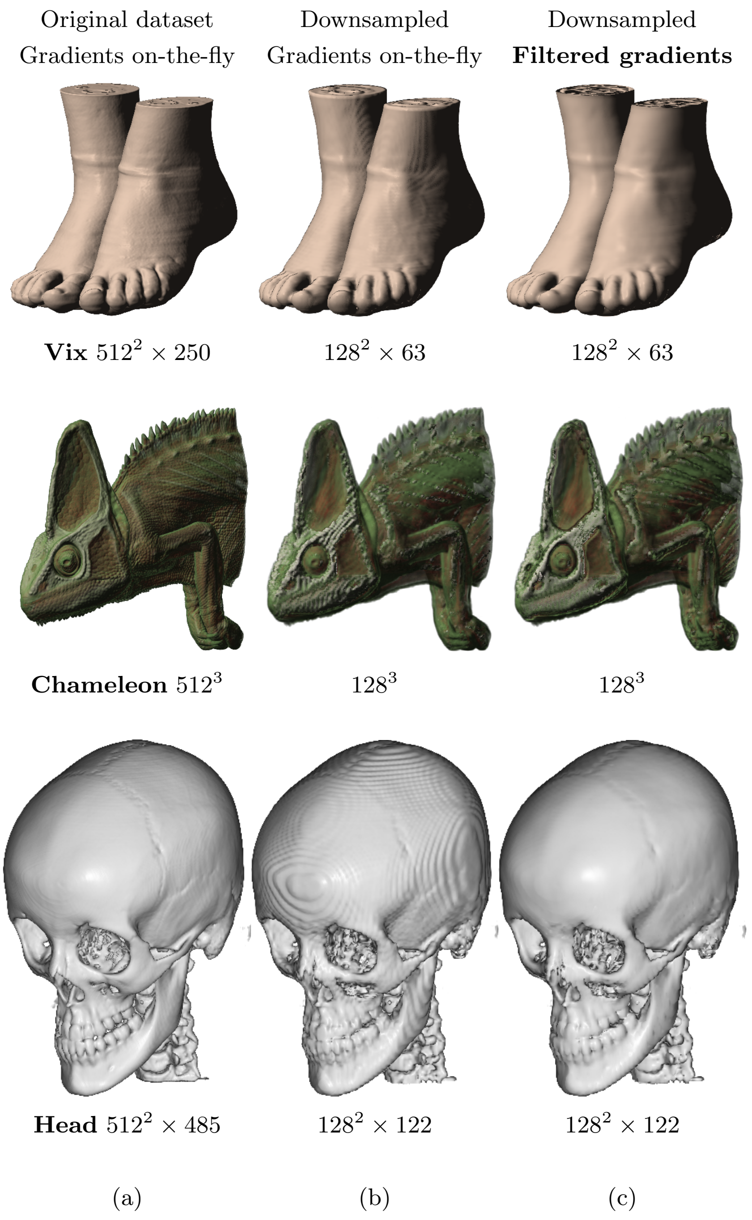


Figure 5.17: Comparison of ray casting images rendered with different gradients. Column (a) shows ray casting images of several datasets at its original resolution (the shading was done using gradients computed on-the-fly). Images in column (b) show a coarse version of the datasets shaded with gradients also computed on-the-fly in the shader (notice the staircase shape of the surfaces). In (c) the same coarse datasets are rendered, using pre-computed, filtered gradients that better preserve the orientation of the original surfaces.

Stage	**Vix**	**Chameleon**	**Head**
Pre-computation	10 s	20 s	18 s
Downsampling	75 s	143 s	134 s
Encoding	62 s	129 s	117 s

Table 5.2: Time (in seconds) used to pre-compute, downsample and encode gradient data of various datasets. The algorithms have been executed in a single CPU thread, traversing the whole sample space of each dataset without further optimization (note that, if necessary, these algorithms could be easily parallelized).

5.6.2 Downsampling and encoding pre-computation time

Table 5.2 shows the times for the pre-computation, downsampling, and encoding of gradient directions using the three datasets in our tests. We have used the central differences approach to pre-compute gradients from the original dataset. The most time consuming, pre-processing stage, is the use of the downsampling filter for the generation of the coarse dataset of gradients, followed by the encoding (using the LUT) plus the quantization of the gradients to pass them to the GPU. The whole process takes a few minutes for the bigger dataset we have tested (Chameleon 512^3), which is an acceptable amount of time for a pre-process. These computations (and the following ones) have been done in a machine with an Intel(R) Core(TM) i7 CPU 930 at 2.80GHz and 8GB of RAM memory. Although the processor has several cores, the calculations done have not been optimized to make use of multi-threading or SIMD instructions.

5.6.3 LUT pre-computation time

In order to perform the encoding of gradients, the LUT for the transformation T must have been previously generated, as solving such difficult operation for all gradients, given the large number of voxels contained in a dataset, is too expensive. This process requires solving the complex system of equations implied by inverting the simple decoding transformation (equation 5.2). We have used the software package R in combination with its module *rootSolve* to compute the transformation T and to store it into a LUT. We have generated LUTs of different sizes and tested their effectiveness to encode pre-computed gradients in terms of encoding time and error. Table 5.3 shows the time taken

LUT size	**Generation time**
32^3	0.55 s
64^3	1.59 s
128^3	6.17 s

Table 5.3: Time needed to generate LUTs of different sizes for the transformation T. Notice that, although these LUTs represent a space in 3D, the time increases approximately by a factor of 4 when the dimension of the LUT increases by a factor of 8. This is due to the fact that the subspace of useful entries in these LUTs is represented by a triangular plane representing only the useful barycentric coordinates (see Figure 5.15).

Encoding method	**Max. error**
No encoding	0.318 deg.
Encoding (LUT 32^3)	0.165 deg.
Encoding (LUT 64^3)	0.165 deg.
Encoding (LUT 128^3)	0.165 deg.

Table 5.4: This table shows the maximum error introduced during the storage of gradients in the 3D texture for a big set of randomly generated gradients. Notice that encoding gradients using our transformation plus quantization scheme produces results twice as good as using a plain quantization without any encoding. Given the monotonic shape of the transformation, even small LUTs (32^3) are enough to obtain the best results (the LUTs are encoding a low-frequency transformation, this is why there is no need for bigger LUTs to achieve good representations of the transformation).

to generate LUTs of three different sizes. Notice that the cost of generating a LUT of 128^3 entries (which is more than needed as explained later) is minimal.

5.6.4 LUT error analysis

We have analyzed the results obtained using a LUT for the barycentric transformation T by completing the whole process of encoding, quantizing, and decoding, for a huge set of randomly generated gradients. Table 5.4 shows the maximum error obtained after performing the test with several storing methods. As we can see, there is no need to use big LUTs (which would be wasting space in main memory), as the lowest resolution LUT used in our tests (32^3) is able to perform the transformation T without exceeding the *biggest hole* angle mentioned in Table 5.1. Although the proposed method for gradient encoding does not provide results that are visually much superior, it achieves a significant improvement in the numerical results that may be useful in other scenarios.

5.7 Conclusions and future work

In volume rendering, the way in which gradients are computed affects in great measure the quality of the shading of coarse datasets in multiresolution structures. Commonly, gradients are evaluated on-the-fly in shader code by accessing neighboring positions. However, the new topology of downsampled datasets provides gradients of worse quality that do not resemble the originals as much as desired, and thus shading shows nondesirable artifacts.

To solve this issue, we have presented two contributions:

- A downsampling filter for pre-computed gradients.
- An encoding/decoding scheme for pre-computed gradient directions.

The proposed downsampling filter for pre-computed gradients provides improved gradients that better match the original dataset gradients such that the previously mentioned artifacts disappear.

Regarding the second contribution, existing algorithms for encoding normal vectors and gradients cannot be used in the context of volume rendering by storing them into a 3D texture. These solutions cited in Chapter 3.3 have serious interpolation issues at the time of sampling values. The presented method to encode gradient directions into a byte precision 3D texture, besides not presenting this problem, maximizes the space of representable directions and reduces the maximum error introduced by the storage format.

In the future, we would like to test the goodness of the proposed encoding scheme beyond the scope of volume rendering, for instance, in combination with triangle meshes.

5.8 Publications

The algorithm explained in this chapter has produced the following publication:

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (June 2017) *Downsampling and Storage of Pre-Computed Gradients. In CEIG 2017: XXVII Spanish Computer Graphics Conference: Sevilla, Spain, (pp. 51-60). European Association for Computer Graphics (Eurographics).* [18]

6 High Quality Visualization of Coarse Datasets

Moving forward through the pipeline, we arrive at the visualization stage. In this phase, Transfer Functions are the entities in charge of converting input data values into output colors. They deserve particular attention in the context of multiresolution visualizations, where other artifacts may appear. This chapter introduces Adaptive Transfer Functions, an algorithm that adapts Transfer Functions to fit downsampled models. It generates improved versions of the original Transfer Function, customized to coarse datasets in the multiresolution hierarchy so that the quality of their renderings is highly improved. The technique is simple and lightweight, and it is suitable, not only to visualize huge models that would not fit in a GPU, but also to render not-so-large models in mobile GPUs, which are less capable than their desktop counterparts.

6.1 Motivation

As already mentioned, multiresolution techniques are broadly used in order to handle the problem of fitting datasets into limited memory or maintaining interactivity when dealing with big volume models. In previous chapters, we have addressed some of the shortcomings of the downsampling process, either regarding the evaluation of the scalar field and the estimation of gradients.

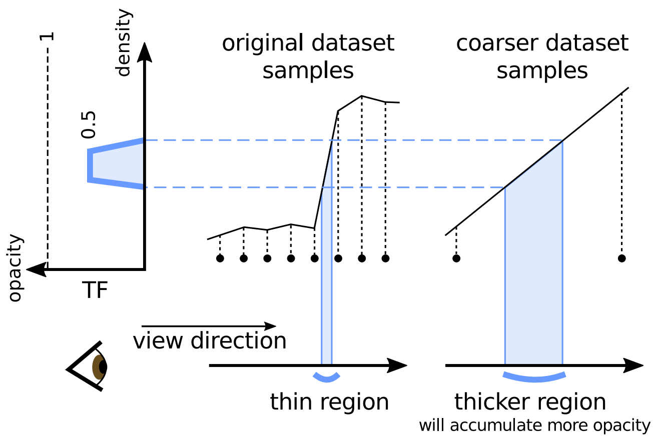


Figure 6.1: Thickening effect on coarse models. Coarse models may not be able to represent high frequencies of the original scalar field (e.g., sudden intensity changes). Transfer Functions designed to visualize thin surfaces (e.g., skin tissue) in the original dataset may detect thicker regions in coarser models, which means that in the case of semi-transparent TFs, coarse datasets will accumulate the colors and opacities of more samples, providing different results than in the original dataset.

Now, if we move further down the volume visualization pipeline, we find Transfer Functions. A TF is a mechanism that allows transforming input dataset values (typically densities) into color and opacity data, in order to provide useful visualizations that highlight interesting features. Furthermore, it is known that Transfer Functions are typically designed by users (usually radiologists), working on high-end machines, to reveal these features of interest in the original, high-resolution dataset. However, in multiresolution visualizations, coarse datasets have suffered a variation in the distribution of density values with respect to the original data. Due to this, the consistency of multiresolution visualizations is affected by the Transfer Function. Renderings of different levels of resolution will have different results if we use the same TF.

High frequencies in the original dataset are not properly represented in coarser representations of the multiresolution hierarchy. A clear example of high frequency, to provide some context, is a sudden change of intensity, as it happens in the boundary between air and skin tissue. As shown in Figure 6.1, TFs designed to visualize thin structures in the original dataset may behave unexpectedly with coarse datasets, because thin structures tend to become thicker with the loss of resolution, and thus, the rendering algorithm may accumulate more colors and opacities, providing undesired results. Figure 6.2 shows an example of this issue.

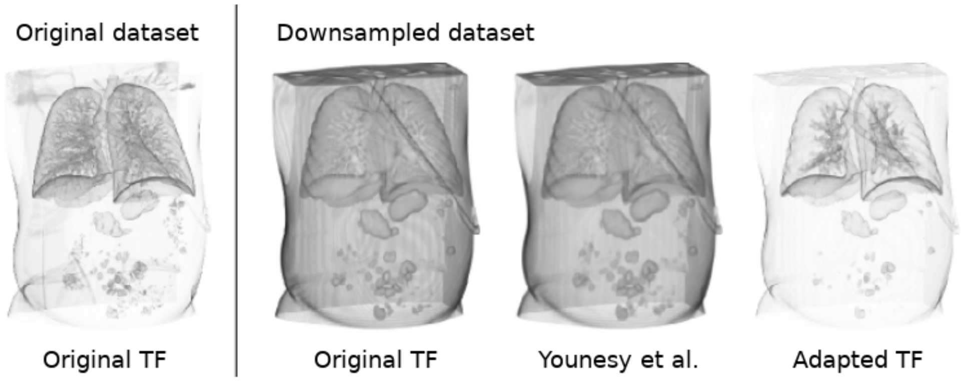


Figure 6.2: Comparison of renderings of the skin tissue. The original dataset (left) rendered using a TF with a low level of transparency. Using the original TF generates more opaque results. The method by Younesy et al. [90] also generates undesirable results in this case. Properly adapting the TF (right) achieves renderings of the coarse dataset which are much more similar to the renderings of the original dataset.

After observing that the scalar field changes across different levels of resolution, it is reasonable to think that different data may require a different TF to visualize the originally desired features. This reasoning led us exploring the possibility of further improving the quality of the renderings of coarse datasets by means of performing changes on the TF besides using the appropriate filters to generate the coarse models of the multiresolution hierarchy.

6.2 Problem addressed

The problem addressed in this chapter is the improvement of the renderings of coarse resolution datasets in a multiresolution hierarchy through the modification of the TF. The computation of representative colors and opacity in coarse datasets has been a subject of study in some previous publications [90, 77]. However, these previous methods do not take into account more restricted hardware, where the amount of graphics memory or processing power can be too limited in order to support demanding algorithms. In this chapter, we perform an analysis of the variation of density values in coarse resolution datasets with respect to the original data. Based on this correlation of densities, we present a novel algorithm that is able to interactively (i.e., allowing interactive TF modification) compute fitted TFs for coarse datasets that achieve a more accurate computation of colors and opacities. The algorithm is simple and lightweight in terms of memory and processing power, allowing very restricted hardware to perform flawlessly even in scenarios where an interactive modification of the Transfer Function is required.

6.3 Theoretical framework

In order to provide a formal context to our problem, let us introduce some definitions which adapt the general definitions already presented in Chapter 2 to the scope of this chapter.

6.3.1 Downsampling of the scalar field

Given a volume dataset defined in a space $D \subset \mathbb{R}^3$, $V(x)$ is a scalar function that computes a density value for points $x \in D$:

$$V(x) : D \subset \mathbb{R}^3 \rightarrow \mathbb{R}.$$

Let us assume that this volume is evenly sampled at N^3 points x_i and stored in a voxel representation of N^3 resolution. For notation convenience, we use V_0 to refer to this original model and $z = V_0(x_i)$ the density value at x_i.

Formally, a multiresolution volume representation is a set of successively coarser resolution models $V_0, V_1 \dots V_n$. We assume a reduction factor of two in each dimension from successive resolutions so that, for $k > 0$, V_k is stored in a 1/8th of memory required for V_{k-1}. For $k > 0$, V_k is usually computed by filtering and downsampling a higher resolution representation. It is evident that the distribution of values in the volume changes among the different levels of resolution.

Without loss of generality and in order to simplify notations, in the rest of this section we will only consider a 1-dimensional scalar field. Downsampling filtering is usually performed through a symmetric weighting function w of finite domain [34]:

$$V_k(x_j) = \sum_i V_0(x_i)\, w(x_j - x_i) \tag{6.1}$$

where, in uniform voxelizations, we can assume that $x_j = j \cdot 2^k h$ and that $x_i = i \cdot h$, with h being a constant spacing.

By considering the voxelization as a discrete representation of a continuous scalar field we can also write:

$$s = V_k(x) = \int_{-\infty}^{\infty} V_0(y)\, w(x-y)\, dy \tag{6.2}$$

where s is a density value obtained from the downsampled level V_k at a specific position x. We assume that values retrieved from any discretized volume are computed by means of a reconstruction filter.

6.3.2 Using TFs with multiresolution datasets

Let $\mathbb{T}F_0$ be a 1D Transfer Function specifically designed to map density values of the original dataset V_0 to output color and opacity, $\mathbb{T}F_0 : \mathbb{R} \to (R, G, B, \alpha)$; and let $I^{\mathbb{T}F_0}(V_0)$ be the image obtained by rendering V_0 using $\mathbb{T}F_0$. Therefore, if V_k is rendered using $\mathbb{T}F_0$, the resulting image $I^{\mathbb{T}F_0}(V_k)$ will differ and lose details from the render of the original volume $I^{\mathbb{T}F_0}(V_0)$ due to the change of values on downsampling. The ideal after downsampling would be to have a new $\mathbb{T}F_k$ with which $I^{\mathbb{T}F_k}(V_k) = I^{\mathbb{T}F_0}(V_0)$, that is, $\mathbb{T}F_k(V_k(x)) = \mathbb{T}F_0(V_0(x))$ for all samples x in D.

Several previous papers [90, 42] have tried to compute the $RGBA$ color C_k for a point x in the downsampled volume V_k by using a color averaging function w_C and defining:

$$C_k(V_k(x)) = \int_{-\infty}^{\infty} \mathbb{T}F_0(V_0(y))\, w_C(x-y)\, dy \tag{6.3}$$

Notice that, in this equation, $C_k(V_k(x))$ is local, being in fact a function of the original densities around x and the original transfer function $\mathbb{T}F_0$.

We observed that the downsampling process from the original dataset V_0 to a lower-resolution V_k (see equation 6.2) can be characterized by a 3D point cloud in the (x, z, s) space, where the x-dimension represents the N^3 voxels of V_0. Any point (x, z, s) of the cloud represents that the computation of the downsampled density $s = V_k(x)$ takes into account the density value $z = V_0(y)$

for y in the neighborhood of x (just observe that we have deliberately removed the second order locality of y in equation 6.2 by ignoring this y value).

We can compact this point cloud in the z-direction by encoding, for each pair (x, s), the occurrences of the z values used to compute $V_k(x)$. We refer to this information as the 2D histograms $H_{x,s}(z)$. Note that $H_{x,s}(z)$ stores the distribution of original density values z within a footprint of x in V_0.

Younesy *et al.* [90], by defining an appropriate weighting of these histograms, transform equation 6.3 into:

$$C_1(V_1(x)) = \int_{-\infty}^{\infty} \mathit{TF}_0(z)\, H_{x,s}(z)\, dz \tag{6.4}$$

(They focus on the case $k = 1$.) The direct use of equation 6.4 requires storing one histogram per voxel in downsampled representations, which is unpractical. Thus, the authors approximate each of those histograms by two values μ and σ (mean and standard deviation) to represent the Gaussian curve that better fits their distribution (note that μ encodes the downsampled voxel density s of $V_k(x)$). This is, in fact, a *projection* of the point cloud in the z direction that allows pre-computing the integral in equation 6.4 into a 2D Transfer Function $\mathit{TF}_k(\mu, \sigma)$.

6.4 Adaptive Transfer Functions

Our approach was inspired by the experimental behavior of the *discretized projection* of the point cloud in the x-direction, which shows a strong $z - s$ correlation with a curve-shaped cloud. For each pair (z, s) we compute the number of sample points x such that $V_0(x) = z$ and $V_k(x) = s$. We refer to this information as the 2D histogram $H^d(z, s)$ (see Figure 6.3). We observed that this projection was clearly showing the before mentioned $z - s$ correlation, even when the locality information on x had disappeared. In other words, the amount of information loss when projecting the point cloud in the x-direction is limited. By using the histograms $H^d(z, s)$ we are able to get rid of the spatial dimension, and we still capture most of the downsampling information. By just changing the projection direction of the point cloud $\{(x, z, s)\}$, we move from local histograms $H_{x,s}$ to our global histograms $H^d(z, s)$.

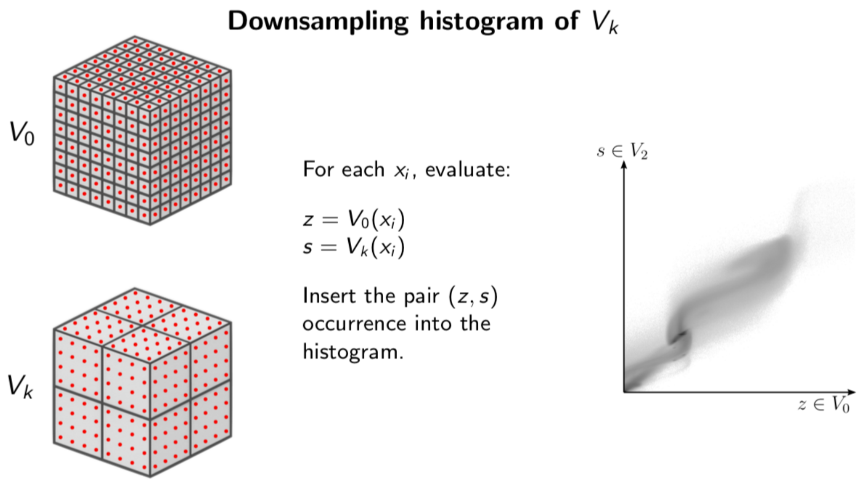

Downsampling histogram of V_k
V_0
V_k
For each x_i, evaluate:
$z = V_0(x_i)$
$s = V_k(x_i)$
Insert the pair (z, s) occurrence into the histogram.
$s \in V_2$
$z \in V_0$

Figure 6.3: The downsampling histogram of V_k is generated by taking pairs of samples from both the downsampled dataset V_k and the full resolution dataset V_0. All pairs of samples are interpreted as 2D coordinates that can be placed in a bin in the 2D histogram. We set the location of the samples in the middle of the voxels of the full resolution model. Thus, the values taken from V_0 can be the contents of each voxel, whereas in V_k a reconstruction filter (e.g., tri-linear interpolation) needs to be used.

We decided to use the downsampling global histograms $H^d(z, s)$ in equation 6.3. The goal is to compute $\mathbb{TF}_k$ so that the resulting $I^{\mathbb{TF}_k}(V_k)$ is as close as possible to $I^{\mathbb{TF}_0}(V_0)$:

$$\mathbb{TF}_k(s) = \int_{-\infty}^{\infty} \mathbb{TF}_0(z_H)\, H_k^d(z_H, s)\, w_H(z_H - \mu(s))\, dz_H \tag{6.5}$$

where w_H is a weighting function that allows us to focus on the z_H-interval that contributes with the highest information, and $\mu(s)$ is the average center point for every s, which can be directly estimated from $H_k^d(z_H, s)$. Now, by imposing that $\mathbb{TF}_k(s)$ should be equal (or as close as possible) to $\mathbb{TF}_0(s)$, we can write:

$$\mathbb{TF}_0(z) = \int_{-\infty}^{\infty} \mathbb{TF}_0(z_H)\, H_k^d(z_H, s)\, w_H(z_H - \mu(s))\, dz_H \tag{6.6}$$

By discretizing equations 6.2 and 6.6 and using the definition of s in equation 6.2, a specific equation is directly obtained for every sample position x.

Observe that equation 6.5 computes $\mathbb{TF}_k(s)$, while equation 6.6 imposes that, for a fixed sample x, $\mathbb{TF}_0(z) = \mathbb{TF}_0(V_0(x))$ should be as close as possible to the resulting value of $\mathbb{TF}_k(s) = \mathbb{TF}_k(V_k(x))$ as evaluated from equation 6.5. The result of equation 6.5 is $\mathbb{TF}_k(s)$, while the unknown in equation 6.6 is the function w_H.

The set of equations 6.6 tries to force all colors and opacities in points of V_k to be identical to their corresponding locations in V_0. Note that this is an overdetermined linear system of equations on the unknown values that the averaging discrete function w_H takes on its finite domain. An optimal solution of this linear system in the least squares sense could be obtained by quadratic programming, by imposing that all discretized values $w_H(i)$ of the weighting function must be positive ($w_H(i) > 0$). However, for efficiency purposes, we have computed the optimum of equation 6.6 in the least squares sense on a restricted domain of positive weighting functions by testing a bi-parametric convex set of functions w_H, as discussed in Section 6.6. Our results show that a Gaussian averaging function w_H has a good behavior in all cases with small Root Mean Square (RMS) and perceptual errors. The following section describes our implementation using these Gaussian averaging functions.

6.5 Fast approximation of Adaptive Transfer Functions

In this section, we present our implementation to approximate the optimal TFs for coarse levels. This is achieved by analyzing the distribution of density values at the higher levels in relation to the finest level, once the multiresolution pyramid has been built.

We precompute $H_k^d(z, s)$ (the 2D-histogram that relates downsampled densities s of V_k and initial densities z of V_0) for each coarse level k. Figure 6.4 shows one of these histograms created by evaluating one value per voxel at the maximum resolution V_0. For the lower resolution level V_k, intensity values are evaluated using trilinear interpolation (mimicking the behavior of GPUs accessing 3D textures). Another example is depicted in Figure 6.5 for the Head dataset. Gray points are in zones where the correspondences take place. Notice how the loss of the original information is reflected by the spreading of the points as the downsampling increases. This indicates that the original TF will not work properly for downsampled levels.

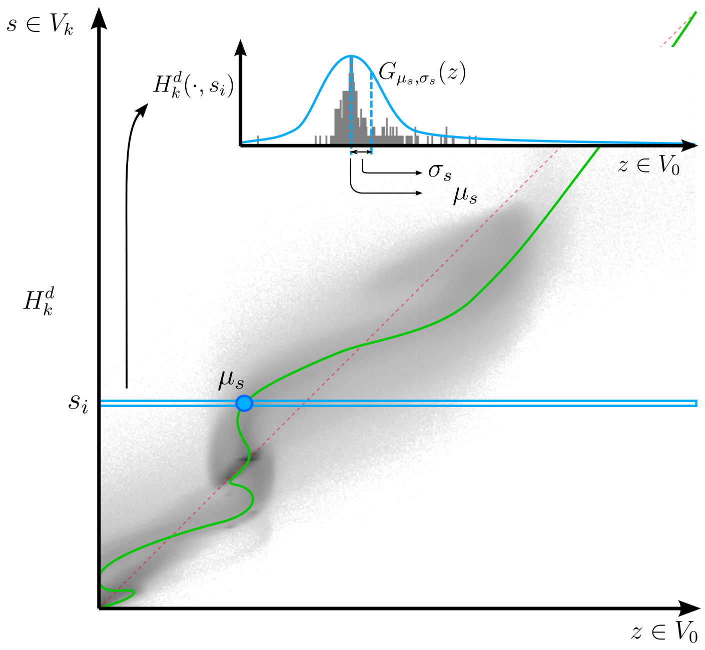
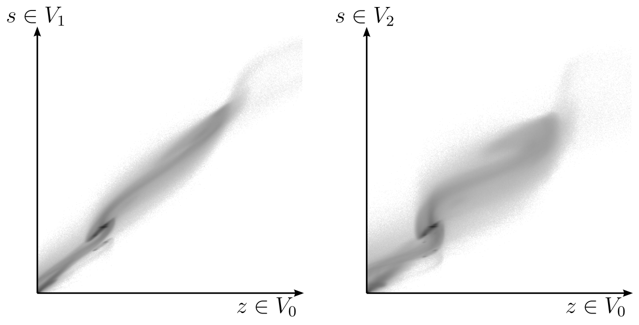
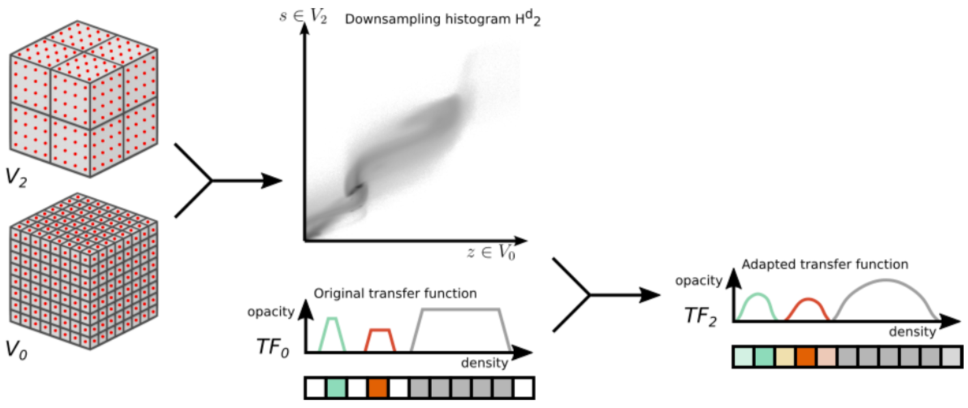


Figure 6.4: Histogram of density correspondences between the original volume V_0 and a downsampled volume V_k in the multiresolution pyramid. Gray points lay in zones where the correspondences take place. The green line is a function of s that approximates a path fitting the mean of each individual row 1D histogram; the row histogram for a given s_i contains the overall information about what density used to be s_i in V_0 before having been downsampled.



(a) Joint histogram H_1^d (b) Joint histogram H_2^d

Figure 6.5: Histograms of correspondences between the full resolution model V_0 and subsampled versions of the Head dataset (see Figure 6.11). (a) shows the correspondences between V_0 and V_1, and (b) shows the correspondences between V_0 and V_2. Note how the values spread increasingly as long as we go to downsampled levels, because of the averaging functions that dilute the details of the voxels. This clearly suggests that using the original $\mathbb{TF}_0$ on V_k will likely be suboptimal.



$s \in V_2$
Downsampling histogram H^d_2
$z \in V_0$
V_2
V_0
Original transfer function
opacity
TF_0
density
Adapted transfer function
opacity
TF_2
density

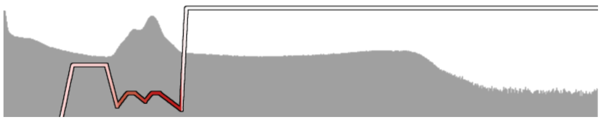
Figure 6.6: Schematic overview of the Adaptive Transfer Functions. The occurrences in each row of the downsampling histogram between V_0 and V_k are used to perform a weighted sum of the color values in the original Transfer Function TF_0, that will provide more accurate colors for TF_k.

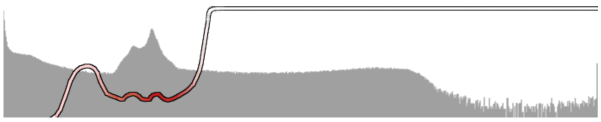
As shown in Figure 6.6, an adapted Transfer Function TF_k is computed by traversing the vertical axis of H_k^d (s values in V_k) and, for each value, averaging the colors of $\mathit{TF}_0(z)$ using the information along the row (z values in V_0):

$$\mathit{TF}_k(s) \leftarrow \frac{1}{K} \sum_z \mathit{TF}_0(z)\, H_k^d(z,s)\, G_{\mu_s,\sigma_s}(z) \qquad \forall s \tag{6.7}$$

Here, G_{μ_s,σ_s} is a Gaussian function centered at μ_s with standard deviation σ_s (see Figure 6.4), and the denominator $K = \sum_z H_k^d(z,s)\, G_{\mu_s,\sigma_s}(z)$, ensures that the weighted sum of colors is normalized. All rows of the histogram are visited, and in our current implementation the algorithm only traverses the values around the mean (we use values of $\pm 3\sigma_s$ around μ_s).

Once our method is applied, fitted Transfer Functions for V_k can be computed very quickly. The resulting adapted TFs for two coarse levels are shown in Figure 6.7. Note how our approach improves the result in comparison to using the original TF as in (b), yielding images that are more similar to the original model in (a). Images (c) and (e) show, as a temperature map, the difference images between the image rendered at full resolution, and the ones obtained with the downsampled models (b) and (d), respectively. The model rendered with the fitted TF used in (d) yields a much better result than the original TF.


(a) Original TF of the full resolution model.


(b) Corrected TF for the first downsampling level.

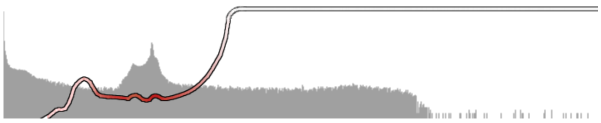

(c) Corrected TF for the second downsampling level.

Figure 6.7: Original Transfer Function (top) and the adapted TFs obtained with our method for two different levels. The bottom one is used for rendering in the Head in Figure 6.11.

6.6 Evaluation and results

This section presents an assessment of the several aspects of the performance of Adaptive Transfer Functions. First, a visual quality analysis exploring the results given by different weighting functions is performed. Then, it discusses the computational requirements of the technique and compares it with well-known competitors. The last subsections show examples on which non-medical models appear, comment about the limitations of our approach and shows some more comparisons among other techniques.

6.6.1 Visual quality analysis

We have analyzed different candidate weighting functions for equation 6.5. To ensure positiveness of this weighting function w_H in the least squares solution of equation 6.6, we have restricted our optimization to a bi-parametric convex set of weighting functions. We use barycentric coordinates on a triangular domain of functions to interpolate among three basis functions: a normalized Gaussian $G(x)$, a constant function $C(x)$ and a triangular function $T(x) = 0.4 \cdot (1 - 0.4 \cdot |x|)$. The bi-parametric weighting function is $w_H(x) = s \cdot C(x) + t \cdot T(x) + (1 - s - t) \cdot G(x)$, with s and t being defined in the interval $[0, 1]$ and

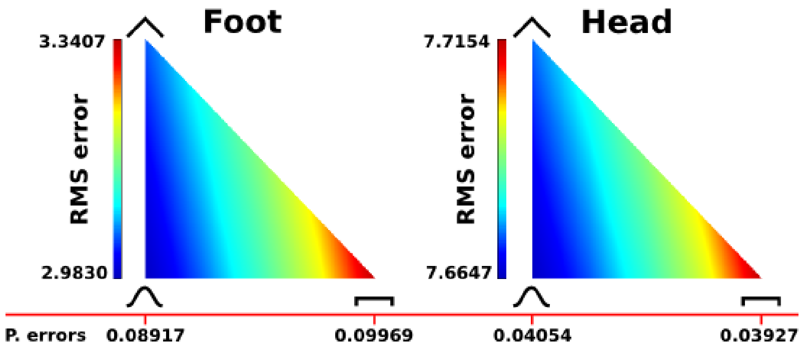


Figure 6.8: RMS analysis of tested weighting functions w_H in equation 6.6. Triangles show the RMS error of all bi-parametric interpolated functions among a Gaussian (bottom-left corner), a Constant (bottom-right) and a Triangular (top) weighting function are shown for two models. The best behavior corresponds to Gaussian weighting, although the color scales show that differences are more important in the Foot dataset. The bottom of the figure also shows the perceptual errors (computed using SSIM measure on 20 views) to further evaluate the effects of both approximations on rendering (see Figure 6.9).

x in the centered interval $[-2.5, 2.5]$. All functions are normalized, having a unit area in this interval. The equations, once normalized, are used to compute the root-mean-square (RMS) error for any w_H defined by a pair of parameters (s, t).

Experiments performed on our test models confirm that the minimum error is obtained at $s = 0$, $t = 0$ in most cases (see Figure 6.8), while in the rest of the cases the resulting error is almost not sensible to (s, t) and to the shape of w_H.

We have also compared the visual quality obtained with the different weighting functions using series of 20 images generated by positioning the camera at the center of all faces of an icosahedron bounding the volume. Each rendering of a downsampled model is compared against the rendering of the original full resolution model using the Structural Similarity (SSIM) index for image quality assessment [85]. The results using Gaussian and Constant weighting functions for equation 6.5 are shown in Figure 6.9, where lower values indicate less error. Errors are higher when using barycentric coordinate interpolation and also using triangle-shaped functions. By analyzing these results, we decided to use Gaussian averaging functions in our implementation. In this way, we reduce RMS and perceptual errors while automatically removing outliers in the histogram of density correspondences. In addition, we have performed a set of experiments to show the advantages of using Adaptive TFs versus the

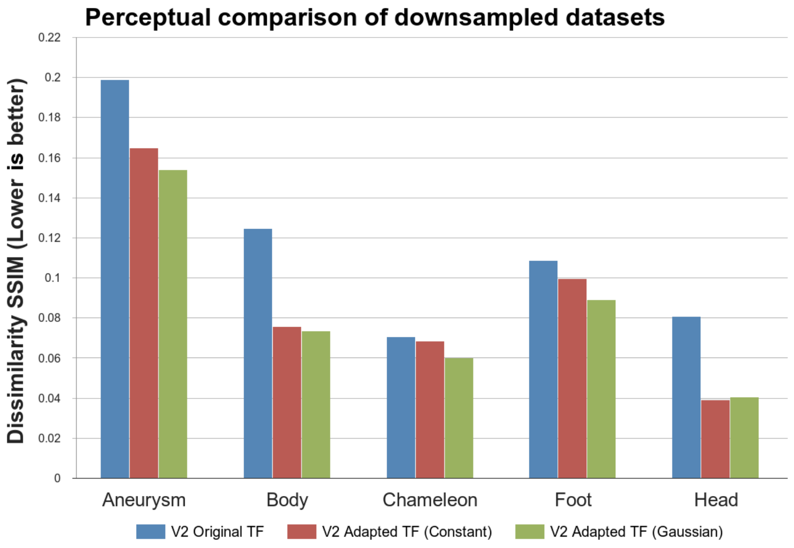


Figure 6.9: Perceptual analysis of two tested weighting functions: Constant and Gaussian. Dissimilarities have been computed with the SSIM perceptually-based metric, as in Figure 6.10. The Gaussian weighting function produces either comparable or better results than the constant function.

original TF with different models. We have used a Quad Core i7 PC and a Core 2 Duo equipped with a GeForce GTX 470 with 1GB of RAM, and GTX 280 with 1GB of RAM, respectively. The resolutions of the models go up to $512^2 \times 1559$ for the Body model, 512^3 for the Head and the Chameleon, and 256^3 for the Foot and Aneurysm. The rendering algorithm is a GPU-based ray casting with pre-integrated classification and on-the-fly gradient computation, and the sampling step is of the size of the voxel (for the corresponding resolution level). In the first PC, the framerate was interactive, and no change was produced with the Adaptive TF. The second PC could only render the large model at 2-3 fps, while our multiresolution rendering with Region of Interest (ROI) is one order of magnitude faster.

In all the examined cases, Adaptive TFs clearly improve the quality of the rendered downsampled model with respect to using the original TF, as shown in Figures 6.11 and 6.12. These images correspond to a two-level simplification from their full resolution models. This data reduction would allow the models shown in this chapter to fit into the GPUs of commodity PCs and most modern tablets and smartphones. We have compared the original models versus three different downsampling levels (see Figure 6.10). Observe that the pairs of bars corresponding to the V_2 and V_3 levels show that adapted TFs are always better

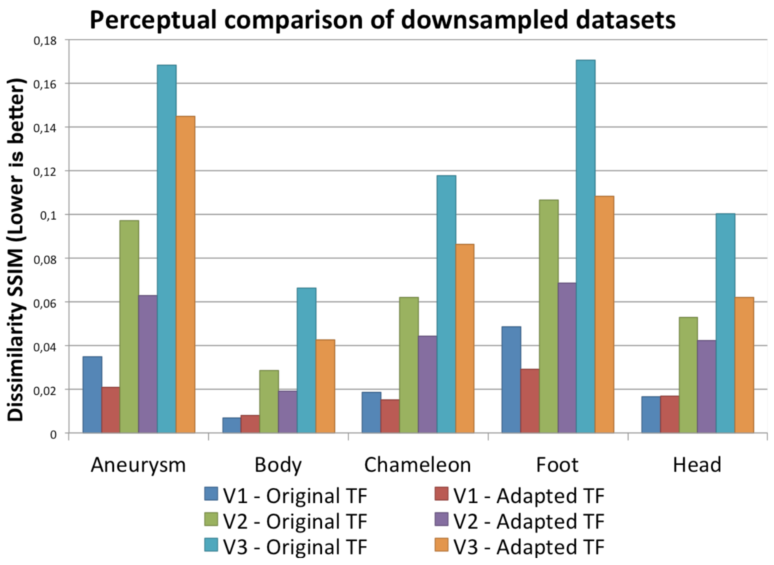
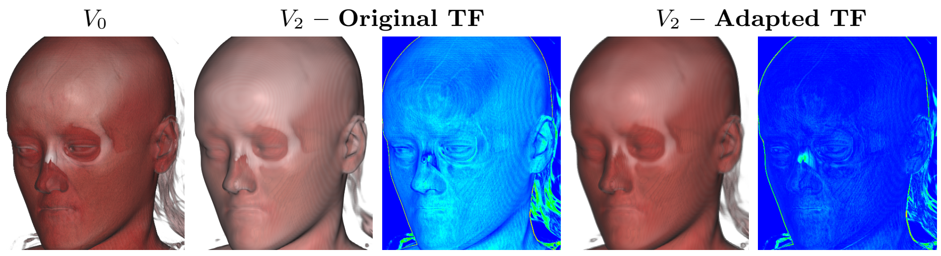


Figure 6.10: Comparison of original models vs. three different downsampling levels (V_1, V_2, and V_3) of the models used in this chapter. The values are computed as the average similarity of 20 regularly spaced views using the SSIM perceptually-based metric [85]. Note that here we use *dissimilarity*, so lower values indicate better performance. Only two models (Body and Head) do not represent an improvement at the first downsampling level, with nonsignificant differences of only 0.0013 and 0.0003.

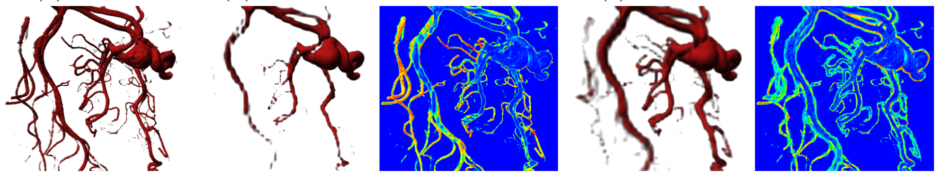
at these levels. The first downsampling level V_1 also improves in three cases while having similar values in the Body and Head models, with negligible differences. Note that, except for two cases where the first level does not improve (only a very small worsening), in all the rest of the cases the differences are noticeable.

6.6.2 Computational requirements

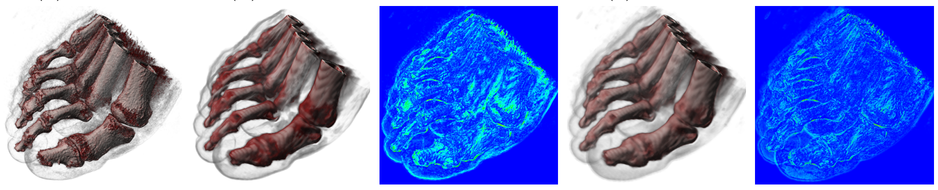
Our method has scarce storage requirements. We need to store an adapted TF at each coarse level, and since it will be recomputed if the original TF changes, we must also keep the downsampling histograms information. Medical data often uses 12 bits per voxel, but in order to reduce storage and computational complexity, we use 8 bits per voxel in downsampled data, as we have seen that this optimization is in fact much milder than the actual subsampling in these kinds of models. Considering this, our technique requires 1 byte per voxel in downsampled levels, plus 256kB (histogram) and $256 \times 4b = 1$kB (adapted TF) per level.



(a) at 512^3 (b) at 128^3 (c) at 128^3

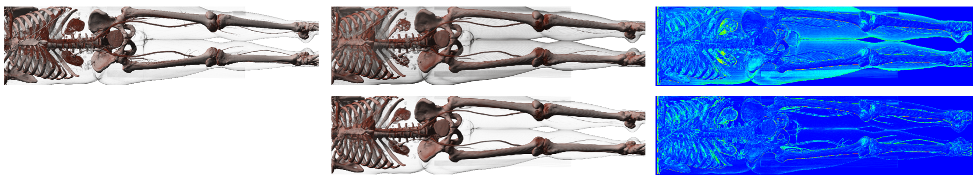


(a) at 256^3 (b) at 64^3 (c) at 64^3



(a) at 256^3 (b) at 64^3 (c) at 64^3

Figure 6.11: Results obtained with our modified TFs for two example datasets (Head and Aneurysm). The leftmost column shows models at full resolution. The second column shows the result of a two-level downsampling without TF change. Our TF adaptation method generates better results as shown in the fourth column. We compare the difference maps from the full resolution model in the third and last columns.



(a) Original ($512^2 \times 1559$) (b) Reduced ($128^2 \times 389$) (c) Difference images

Figure 6.12: The results of our method applied to a simplification of two levels of a $512^2 \times 1559$ model. The images show the improvement that is very noticeable (zooming in will reveal more details). The top row contains the original model (left), and the downsampling without TF adaptation. The bottom row uses an adapted TF. Note how different structures such as the kidneys are better preserved and the overall color of the image is highly improved. The images on the right illustrate the differences between the low-resolution model and the original one.

Method	**Dataset resolution**	**Total size**	**Overhead vs** $256^3 + 128^3$
[90]	512^3	226MB	75.5MB
[77]	512^3	241MB	90MB
Ours	512^3	151MB	256kB+1kB

Table 6.1: Storage requirements comparison for a multiresolution of the 8 bits per voxel 512^3 Shepp-Logan model with two levels of downsampling. The original downsampling ($256^3 + 128^3$) requires 151MB. The method by Younesy *et al.* [90] requires 4 bytes per voxel: one byte for the average μ, one byte for the standard deviation σ and two bytes for the gradient. The approach by Sicat *et al.* [77] uses a sparse structure. The values here are the ones declared by the authors applied for the 512^3 Shepp-Logan model.

Whenever the user changes the original TF, we need to adapt the TFs of the downsampled levels. This entire process takes fractions of a second (less than 0.01 seconds in our tests using 8 bits per voxel), so it is performed interactively. The achieved computation time is an insignificant amount of time, and it is definitely much faster than the time required by other, more complex techniques. A nice feature of Adaptive TFs is that they do not require special purpose modification of the rendering algorithm. Thus, the framerates do not decay, while still improving the quality.

We compare our requirements with the one specified by Younesy *et al.* [90] and Sicat *et al.* [77], as declared by the authors in their respective publications. In the first case they require 4 bytes per voxel, and in the second case, they store a sparse histogram whose size may vary depending on the model data. Table 6.1 shows a comparison of the storage requirements for the Shepp-Logan model of 512^3 voxels. As it can be seen, our method clearly compares favorably against the others.

6.6.3 Behavior on non-medical models and other examples

Although our research has been focused on medical data, Adaptive TFs can also be successfully applied to other volumetric models. We show an example in Figure 6.13 where the Nucleon and the Chameleon datasets are shown. Note that even with an aggressive downsampling such as the one in the Nucleon, where the original model is only of 41^3, and thus the 2-level simplified version

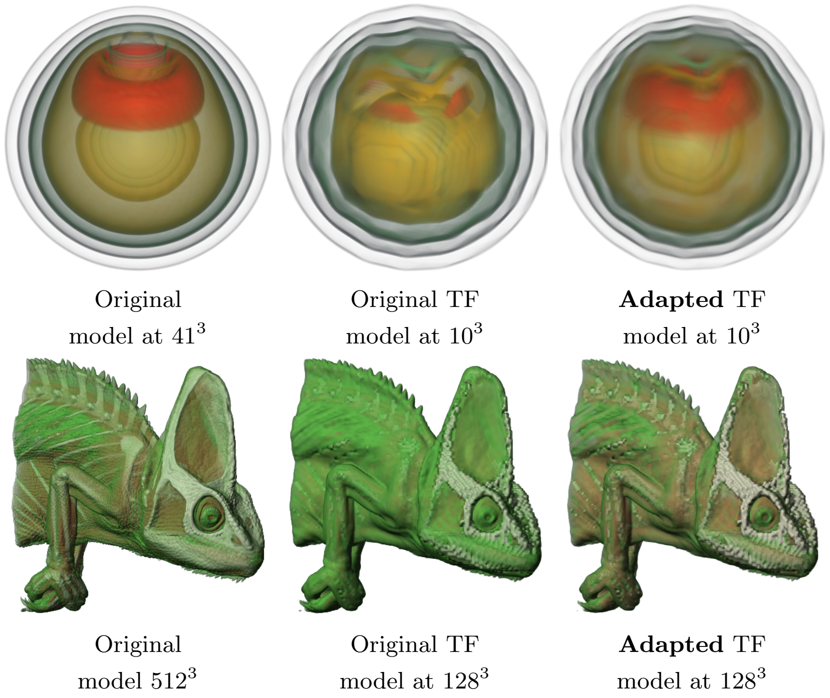

Original model at 41^3
Original TF model at 10^3
Adapted TF model at 10^3
Original model 512^3
Original TF model at 128^3
Adapted TF model at 128^3

Figure 6.13: Using adapted TFs for non-medical models, the Nucleon and the Chameleon, with two levels of downsampling, also achieves good results. The Nucleon dataset is very small; it is only used to illustrate that even with an aggressive downsampled version of 10^3, our fitted TF is able to recover quite a lot of information from the original model.

is only of 10^3, the information we are able to preserve is quite important. We can see it in the perceptual-based comparison where dissimilarity (computed using 20 views as in the previous chart) for the Nucleon dataset is on average 0.216 when we compare the 2-level downsampled model with the original one while when using our Adaptive TF it is reduced to 0.078. For the chameleon, the 2-level downsampled comparison yields dissimilarities of 0.062 with the original TF and 0.044 with our Adaptive TF.

Our approach can be used to render a simplified version of the model while showing the model at maximum resolution inside a user-defined ROI (see Figure 6.14). Note that the transition between the two resolutions is not perceived.

The results obtained outdo the performance of commonly applied algorithms, since they allow us to *recover* information that was lost during the downsampling at a low cost, both in terms of memory and speed.

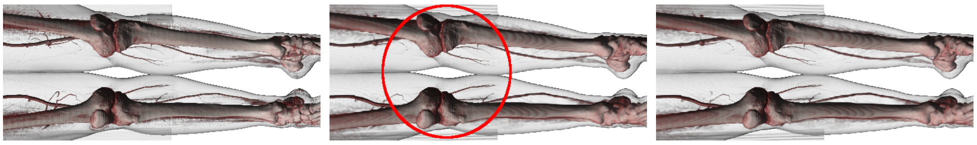

(a) V_0 (b) V_2 with ROI (boundary shown) (c) V_2 with ROI (boundary hidden)

Figure 6.14: Part of the model in Figure 6.12 at the original resolution (a) and two levels of subdivision with the proposed algorithm (b) and (c). In (b) and (c), the simplified model V_2 is shown outside the Region of Interest (ROI), while the ROI shows the original model V_0. The ROI boundary is drawn in (b) for the purpose of comparison with (c).

6.6.4 Limitations

A *limitation* of our technique is its inherent global character. Different neighborhoods of the voxels in the original dataset may be downsampled to the same value on the lower resolution model. Given this fact, although we achieved quite successful results in the models we tested, our TF adaption technique is not able to capture little and thin features. Its improvements will likely be limited in scenarios with highly heterogeneous models, and in the worst case, different textured structures could be equally colored, although it is unlikely that typical medical models will behave this way. To that aim, the feature-preserving downsampling filter presented in Chapter 4 may be worth being introduced in the visualization pipeline. As they are not exclusive, the combination of both techniques, *Adaptive Transfer Functions* and the presented feature-preserving downsampling filter, is a solution that achieves renderings of low-resolution datasets with an accurate computation of colors and opacities, and preservation of fine details.

6.6.5 Comparison with other techniques

We have performed comparisons of Adaptive Transfer Functions with other techniques that supposedly generate good quality coarse representations of the original data in terms of the evaluated colors and opacities.

First, we implemented the method proposed by Younesy et al. [90]. As explained in Section 6.3.2, their technique generates coarse representations that store extra per-voxel information. More precisely, at each voxel of the coarse dataset, they store a mean density value and the standard deviation of densities in its footprint in the original resolution dataset. With these two values and a precomputed 2D Transfer Function, they are able to obtain a more accurate color contribution. Figure 6.15 shows a comparison of several downsampled models rendered with different techniques to obtain the contribution of colors and opacities. The worse results are obtained using the original Transfer Function to render the coarse dataset. Either using Adaptive Transfer Functions or Younesy's method, the evaluated colors and opacities exhibit a great improvement with respect to the previous case. Younesy et al., with their method, are able to preserve some thin structures (e.g., the catheter in the Thorax model) thanks to its higher degree of locality. However, their implementation has higher pre-processing and memory size requirements, which make this technique less practical for systems with more restricted capabilities. Our method, Adaptive Transfer Functions, achieves comparable results with much fewer requirements.

We also implemented another downsampling method, based on the previous pre-classification of scalar data, and later downsampling of RGBA data. As stated in Kraus and Bürger's paper [42], downsampling of scalar volume data for post-classification volume rendering is considerably more difficult and less well understood than downsampling of RGBA volume data. That is due, as said, because Transfer Functions are highly non-linear, and as a result, the relationship between mutations in the scalar field and the final renderings is also non-linear. According to this statement, the renderings shown in Figure 6.16 tend to show better results in the case of downsampling of RGBA data. The technique presented in this chapter also exhibits good behavior, but more aggressive simplifications would incur stronger visual penalties. Downsampling of RGBA data, although robust and more predictable, comes with important drawbacks. Mainly, it needs four times as much memory space as the method we propose, which makes RGBA data not suitable for commodity and mobile devices. Moreover, the computation of color (actually, the whole multiresolution hierarchy) is generated at a pre-processing stage, which means that the TF cannot be changed interactively, as the whole multiresolution hierarchy should be rebuild.

For these reasons, the competitors we compare against may be good options for more powerful hardware such as desktop computers, but actually not good candidates to be implemented in many less powerful, portable devices.

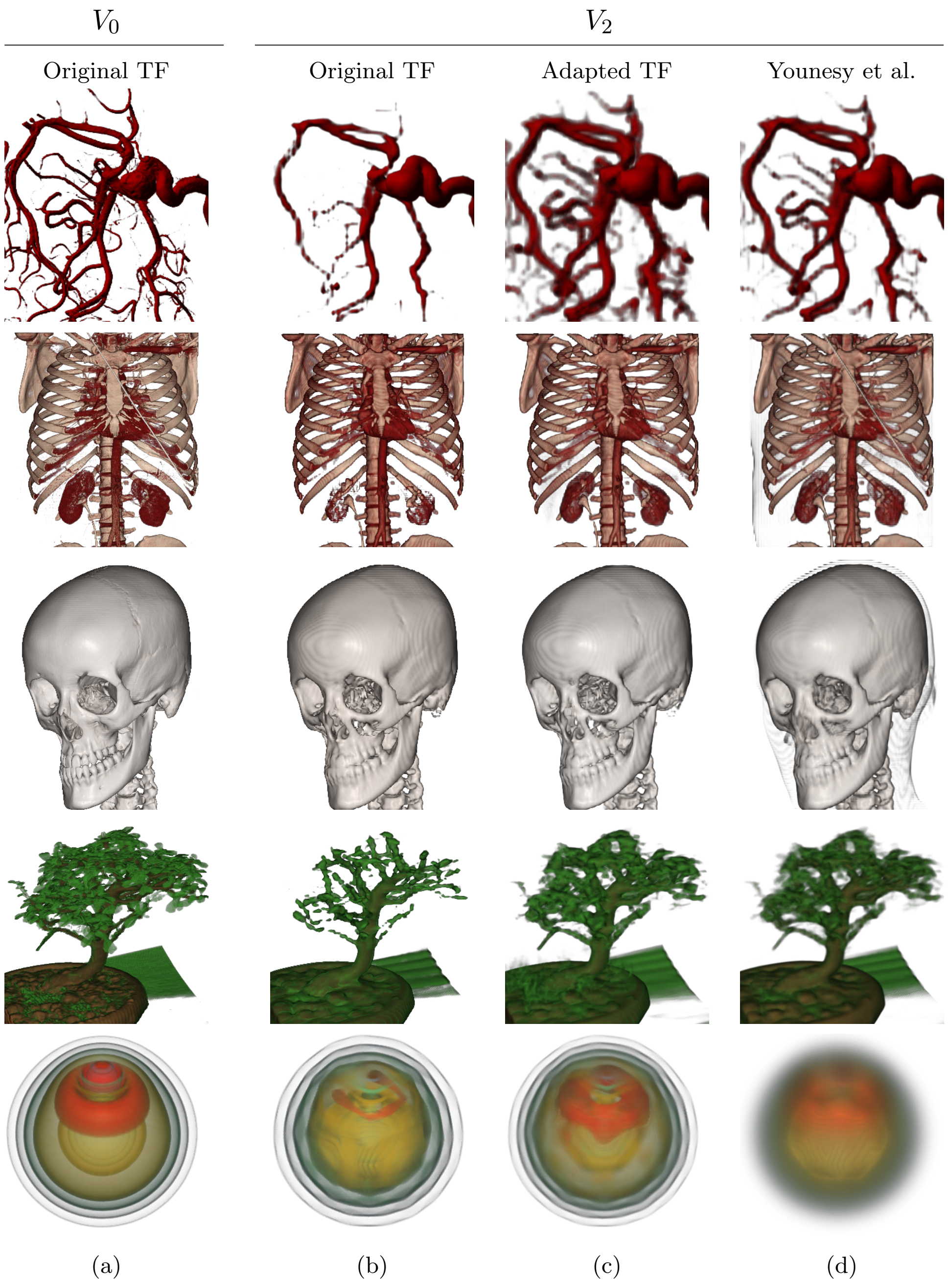


Figure 6.15: Comparison of different levels of resolution of several models whose colors and opacities were obtained using different methods. Column (b) shows the downsampled models (V_2) rendered with the original Transfer Function, the same one used in column (a), for the original dataset V_0. In column (c), coarse datasets V_2 are rendered with our Adaptive Transfer Functions. Column (d) shows renderings of V_2 using the technique by Younesy et al. [90]. Our method (c) achieves quality results comparable to (d), requiring less computational resources.

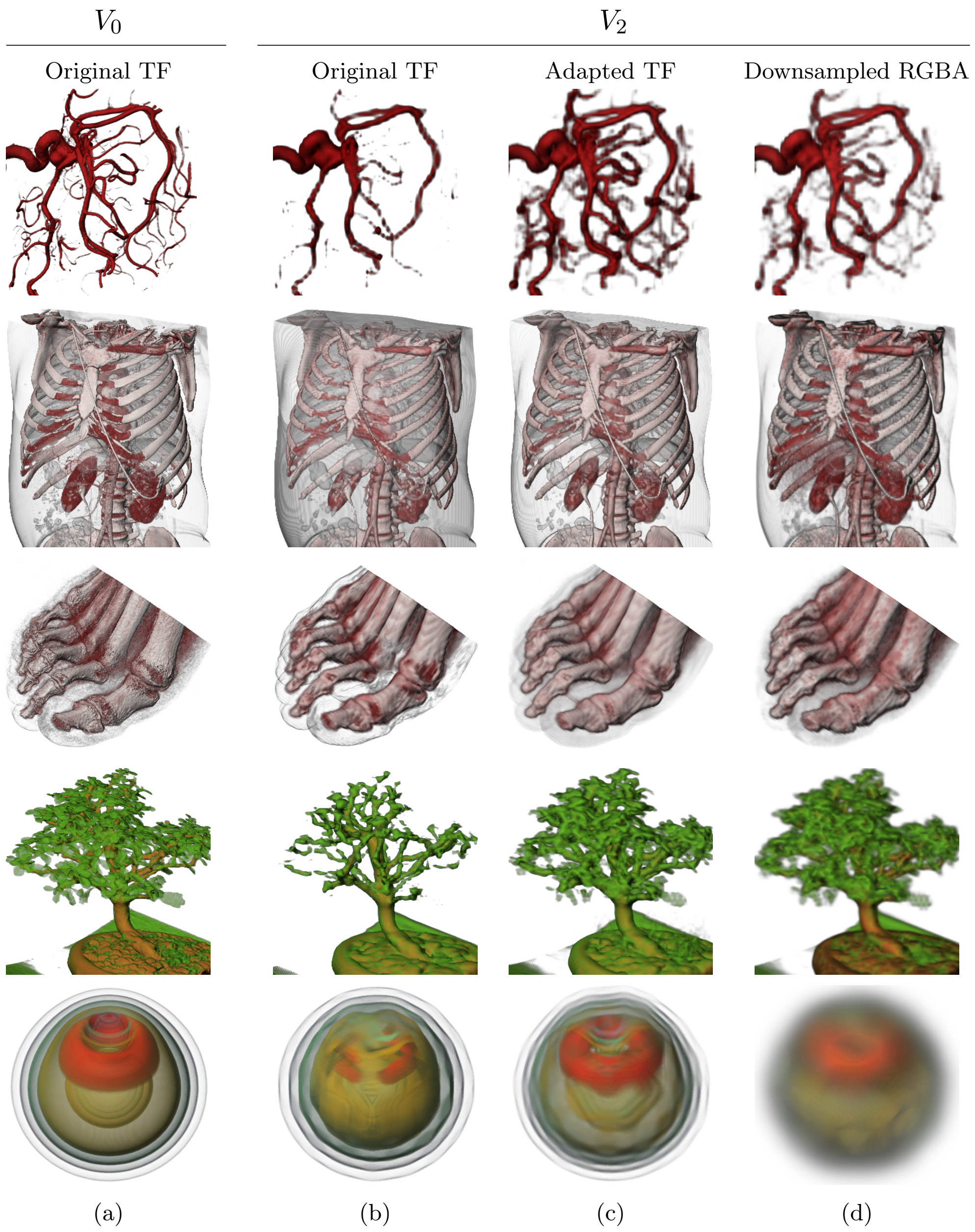


Figure 6.16: Comparison of different levels of resolution of several models whose colors and opacities were obtained using different methods. Column (b) shows the downsampled models (V_2) rendered with the original Transfer Function, the same one used in column (a), for the original dataset V_0. In column (c), coarse datasets V_2 are rendered with our Adaptive Transfer Functions. Column (d) shows renderings of RGBA datasets V_2 that were downsampled after the pre-classification of V_0. The last case tends to provide a higher fidelity as compared to V_0. However, it needs a pre-computation of the RGBA multiresolution hierarchy, the amount of memory needed is four times the required by our approach, and does not allow an interactive modification of the TF.

6.7 Conclusions and future work

In this chapter, we have addressed the problem of the correct computation of colors and opacities in the visualization of coarser levels of multiresolution datasets through Adaptive Transfer Functions. This technique takes into account how scalar field values have changed during the downsampling to compute, for each coarse level, a joint histogram of the correlation between the original density values and the downsampled ones. With these histograms, and starting from the original Transfer Function, it creates fitted Transfer Functions for coarse datasets so that the quality of those renderings is highly improved with respect to always using the original Transfer Function. Unlike other approaches [90, 77], it does not store extra local information at the voxels level, so its memory footprint is very small. It is a global approach, and as such, it succeeds in preserving global features such as an improved estimation of the overall color and opacity.

This technique has four main advantages:

1. Storage costs are negligible (256+1kB for each coarse model).
2. The rendering is not affected using pre-integrated or post-classification, and the gradient -so shading- may be computed on-the-fly (thus, it can be combined with other data storage management methods).
3. The computation requires no manual parameter setting and is fully automatic.
4. It is performed interactively, allowing interactive modification of the Transfer Function.

It is important to note that the method is orthogonal to any downsampling method, so it is not restricted to a subset of the original points, and so it may use any downsampling filter. It may also be seamlessly combined with any compression technique that generates density values. In this case, the decompressed model would be the one used to create the histograms, and during rendering time, it is then just necessary to substitute the TF fetch for a function that fetches the newly created adapted TFs. This approach allows to automatically obtain coarser models that can be used in modest GPU environments and is a good candidate technique for mobile rendering. Obviously, its combination with off-line or bricking techniques is straightforward.

For the preservation of thinner and smaller features, the adapted TF may still not be enough, as a higher degree of locality would be needed. We think that exploring other criteria for the construction of the downsampling histogram could provide valuable information about the downsampling process and would help to recover a certain degree of locality. Right now, just the variation of densities among levels of resolution are used to generate the downsampling histograms used by the Adaptive Transfer Functions algorithm. Other alternatives could include using the gradient magnitude, or the distribution of densities in the neighbourhood of the low-resolution voxels.

Anyway, with the aim of preserving fine details, the feature preserving downsampling filter presented in Chapter 4 is an ideal candidate to include in the volume rendering pipeline. The combination of both techniques, the feature-preserving downsampling filter, and the Adaptive Transfer Functions, produces outstanding results.

6.8 Publications

The technique described in this chapter has produced the following publication:

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Frederic Pérez, Pere-Pau Vázquez. (June 2016) *Adaptive Transfer Functions. Improved Multiresolution Visualization of Medical Models. The Visual Computer, 32(6-8), 835-845.* [16]

7 Interactive Rendering on Mobile Devices

The previous chapters have proposed improvements to some known issues and artifacts in prior stages of the visualization pipeline. This chapter approaches the last stage of the visualization pipeline: the rendering algorithm. Here, we explore the rendering of medium to large volumetric models on mobile and low-performance devices in general. To do so, we propose a multiresolution framework based on the use of an incremental rendering scheme that provides a smooth transition between low-resolution and high-resolution models. Then, we present two new progressive GPU ray casting techniques that fit this scheme so that the achieved visualizations are able to obtain interactive frame rates and high-quality results for models that not long ago were only supported by desktop computers.

7.1 Motivation

In the last years, thanks to its ubiquity and increasing computational power, smartphones, tablets and mobile devices in general, are more and more suitable for applications that require high-quality visualization of volume data in real time, being the clinical practice one of the most important fields in such scenario. Unfortunately, despite the fact that their capabilities, in terms of computational power, visual quality and storage capacity are undeniably raising, there are certain tasks such as the interactive inspection and high-quality visualization of high-resolution medical datasets which still entails a challenging problem in this kind of hardware.

Previous experiments [3] have shown that, even though big models might fit into such GPUs memory, the rendering performance achieved by mobile devices is still not enough. Usually, the visualization of models with larger resolutions ($> 512^2$) that still fit in the graphics memory of mobile devices, achieves low frame rates which are far from being interactive and prevent the user from experiencing a smooth inspection of the model.

When the calculations needed to generate a high-resolution output image exceed an amount of time that allows interactive visualizations, a viable solution is splitting the whole process into several batches whose computation time fit in a reasonable frame time, thereby completing the process incrementally. There are several techniques aimed at the incremental generation of volume renderings, which allow separating the workload in several time steps (see Section 3.5.3 for further details on some techniques using this approach). However, none of the previous work is aimed at the efficient design of an incremental technique suitable for mobile devices.

7.2 Problem addressed

This chapter focuses on solving the issue of high-resolution renderings in mobile devices while maintaining interactivity. Because mobile devices have hardware specifications much more restrictive than desktop machines, special considerations have to be taken into account to visualize medium to large datasets on these devices. For that purpose, we have identified two separated use cases in the application workflow: interaction (to find an interesting view) and inspection (to observe the selected view in detail). We have proceeded differently depending on the use case, as the interaction phase mainly requires the application to be responsive to the input events, whereas, in the inspection phase, the most important requisite is a high-quality visualization.

We have approached the problem described above with use of multiresolution techniques for volume rendering on mobile devices. Our framework uses a low-resolution model during user interaction, and a high-resolution dataset for quality visualization when the camera stops. The solution we propose employs progressive GPU-aided ray casting algorithms to generate the high-resolution renderings. By progressive we mean that the whole high-resolution rendering, when required, is split into parts and distributed over subsequent frames. This strategy prevents blocking and provides a higher degree of interactivity. Specifically, the main contributions presented in this chapter are:

- A strategy pattern for incremental rendering that provides a smooth transition from the low-resolution visualization to the high-resolution visualization, preventing blocking to avoid undesirable application aborting and allowing for smooth interactions at any time.

- The proposal of two new progressive ray casting methods that fulfill the goals mentioned above, and their analysis, and a comparison with other existing techniques.

7.3 Framework overview

We use GPU-aided ray casting to perform direct volume rendering, as it is the state of the art technique for the task [34]. For details on the ray casting algorithm, the reader may refer to Chapter 2. Ray casting is easy to implement and performs highly optimally in graphics chips thanks to the possibility to directly map the rays of the volume rendering integral to individual shader processing units. However, the visualization of large datasets involves costly computations of the ray integral, which implies a bottleneck in the fragment shader performing that calculation.

Our implementation of the ray casting algorithm uses several existing methods that help to improve the visual quality of the final renderings and the performance of the visualization process (see also Chapter 2). For instance, stochastic jittering and pre-integrated Transfer Functions [23] are used in order to avoid undesired wood-grain artifacts without sacrificing performance. We also have incorporated acceleration techniques such as Empty Space Skipping (ESS) using close-fitting proxy geometry, and Early Ray Termination (ERT) whenever possible. In addition, we perform downsampling to achieve a low-resolution dataset that allows interactive exploration, and also whenever the original resolution dataset does not fit the GPU memory. Regarding the contributions of this thesis, we use the feature-preserving downsampling filter described in Chapter 4, which is able to preserve important features that are typically lost during the downsampling process. Finally, we use Adaptive Transfer Functions (see Chapter 6) to visualize the coarser levels with higher accuracy.

The standard way used by medical experts to inspect medical images is based on orthographic projections. For this reason, we use orthographic cameras to generate our renders.

Since our goal involves implementing a scalable system that is able to perform interactive high-resolution ray casting of large models, we propose a framework based on multiresolution. Our solution uses a lower resolution dataset for the visualization while the user is exploring the model, which ensures interactive frame rates, and a high-resolution dataset along with a progressive refinement algorithm for high-quality rendering of the desired regions of interest after each user interaction.

The usage of a progressive rendering algorithm ensures that, by splitting the ray casting into several frames, the control of execution is returned back to the application loop more frequently. This way, it cannot stall for long periods of time, allowing the user to start new interactions at any time, even before the progressive render has finished (thus canceling the process).

Based on this general strategy, we propose two different approaches, depicted in Figures 7.1 and 7.3. Both share the same structure: during user interaction, rendering is performed using low-resolution ray casting (top row). Every time the user stops at a certain view, the progressive high-resolution ray casting starts so that the static image of the selected view evolves smoothly from the low-resolution ray casting result to the full resolution image.

The main differences between both strategies are the way the high-resolution images are produced. In one case, the final image is obtained by rendering the high-resolution dataset in separated slabs in front-to-back order (we call this technique Front-to-back Slabs, or *FBSlabs*). The second strategy, on the contrary, splits the viewport into several tiles and sorts them by cost in order to group them into batches of a similar cost that can be efficiently rendered at each frame until a certain time budget is reached (we refer to this technique as Sorted Tiles, or *STiles*).

7.4 Progressive ray casting strategies

In this section, we present details on each of the two proposals for incremental rendering: *FBSlabs* and *STiles*. On the one hand, *FBSlabs* (which stands for *front-to-back slabs*) is an incremental ray casting algorithm based on object space partitioning, where thin slabs of the scalar field are rendered in subsequent frames in front-to-back order. On the other hand, *STiles* (which stands for *sorted tiles*) is another incremental ray casting algorithm, based on screen space partitioning. It splits the screen space into square blocks of pixels with an assigned a cost, and the high-resolution rendering is incrementally computed in subsequent frames, rendering batches of tiles by cost order. The next subsections go into more detail on these algorithms.

7.4.1 *FBSlabs* (front-to-back slabs)

This progressive algorithm splits each ray into several segments of a fixed length and then starts rendering those segments in front to back order over subsequent frames after the user finishes interacting. The algorithm renders the incremental high-resolution results into a texture T_{high}, the low-resolution results into another texture T_{low} and finally composites both textures to achieve the final image at each frame. These are the main steps followed by this algorithm:

1. **Low Resolution Ray Casting (during user interaction)**
   - The ray casting color is stored in a 2D texture
2. **Progressive High-Resolution Ray Casting**
   (a) A 2D texture T_{high} is cleared
   (b) The first sampling position for each ray (one per viewport pixel) is placed at the entry point on the proxy geometry bounding the volume model
   (c) A fixed number of ray casting steps are performed advancing over each ray (rendering a non-regular slab perpendicular to the viewing direction), and the resulting color is composited with the previous high-resolution partial result in T_{high}
   (d) The remaining part of the volume is rendered with low-resolution ray casting, starting at the sample position where the previous step finished, and then stored in T_{low}
   (e) The current image is generated by compositing T_{high} on top of T_{low} with alpha blending
   (f) In the next frame, a frame counter is increased, and the process resumes ray casting from (c) until the sampling positions exceed the volume boundaries

In Figure 7.1, step 1 indicates that the low-resolution rendering is generated by a standard ray casting algorithm, with no modifications, into a 2D texture T_{low}.

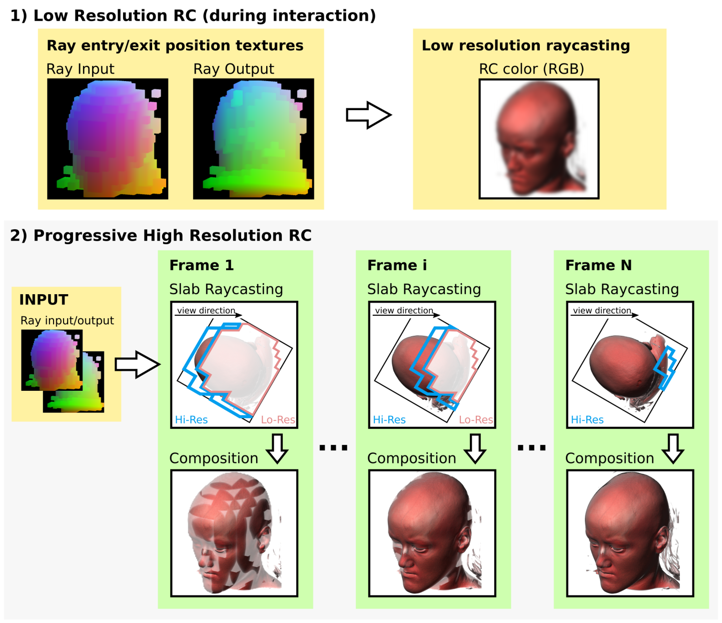


Figure 7.1: Schematic overview of the *FBSlabs* algorithm. Step 1) depicts the initial low-resolution standard ray casting performed while the user is moving the camera. Each time the interaction stops, in step 2), the high-resolution image is incrementally composited by rendering slabs in front-to-back order, one at a time every frame. Then, at each frame, this high-resolution image is composited on top of the remaining part of the model rendered at low resolution.

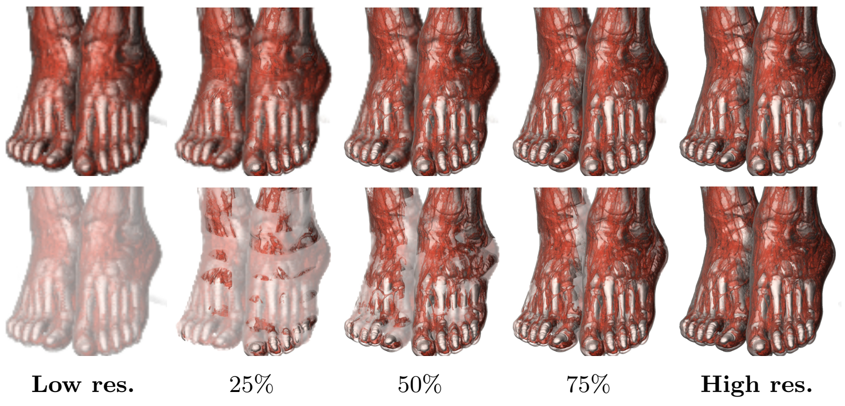


Figure 7.2: Vix dataset ($512^2 \times 250$ high res., $128^2 \times 63$ low res.). This sequence of images shows the transition process from the low-resolution to the high-resolution ray castings obtained by the *FBSlabs* method. The top row shows the renderings as shown in the application, whereas in the bottom row, the low-resolution part of the same images is lightened in order to reveal the updated portions of the image more clearly.

Then, in step 2 of Figure 7.1, a chain of partial ray castings is performed in separated slabs to render the high-resolution dataset progressively. A 2D texture T_{high} is used to store the progressive state of the high-resolution render. To start the process, in the first frame after the user interaction finished, the initial segments of all rays emerging from the viewport pixels are rendered in T_{high}. Those ray segments start at the entry points on the proxy geometry and perform a fixed number of samples ($N = 40$ in our case) in each ray casting frame, making each slab have a fixed thickness. During the next frames, the same slab rendering process is repeated. At each frame, in order to resume the high-resolution ray casting where the previous frame finished, we only need to know the current frame counter (number of frames since the progressive ray casting started), as each slab is rendered with a fixed number of samples and a constant sampling space between them. Note that the camera is configured to perform an orthographic projection of the scene, as commonly used in medicine. This way the generated slabs remain planar as they originated from the proxy geometry. A perspective projection could be used otherwise without causing any trouble, this way leading to pseudo-spherical slabs as they get far from the starting point at the proxy geometry. The blending state is configured to add color in a front to back order in order to update T_{high} with each rendered slab. In T_{low}, the remaining part of the ray casting is computed at low resolution, which implies almost no penalty in time. At each frame, the resulting partial image T_{high} is composited over the low-resolution image T_{low} using alpha blending. The high-resolution ray casting is completely finished whenever all the ray segments rendered exit the proxy geometry. We conservatively approximate this moment by repeating this iterative process until the computed rays are longer than the diagonal of the volume bounding box. Figure 7.2-top shows the transition effect of this technique (in Figure 7.2-bottom, color is modified to better perceive the boundary between the low-resolution and the high-resolution rendered parts).

7.4.2 *STiles* (sorted tiles)

This progressive ray casting algorithm first decomposes the high-resolution image space into square blocks of pixels (tiles) and then renders them progressively over subsequent frames (see Figure 7.3). The rationale behind this method comes from the tile-based behavior of the GPU rasterizer and cache usage. Analogously to *FBSlabs*, the low-resolution rendering is stored into a low-resolution texture T_{low} and the high-resolution results are incrementally rendered into a high-resolution texture T_{high}. The algorithm pipeline proceeds

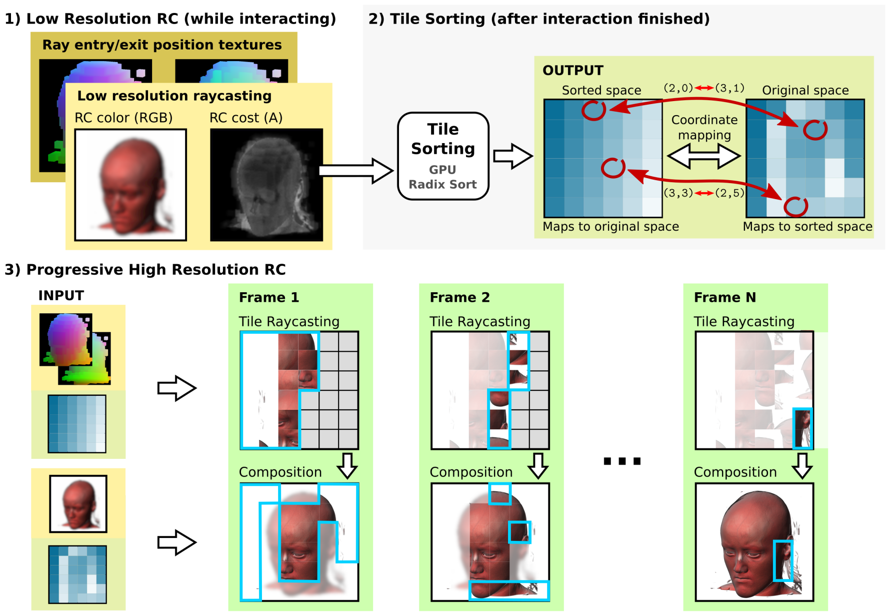


Figure 7.3: Schematic overview of the *STiles* algorithm. Step 1) depicts the initial low-resolution standard ray casting performed while the user is moving the camera (the ray cost hint is stored in the alpha channel). Each time the interaction stops, in step 2), the screen space is split into tiles and sorted according to this ray cost hint, and two mapping textures that are able to convert from one space to another are generated. In step 3), the incremental rendering proceeds frame by frame, rendering tiles in order and compositing the final image by selecting pixels either from the low-resolution texture or from the high-resolution tiled texture.

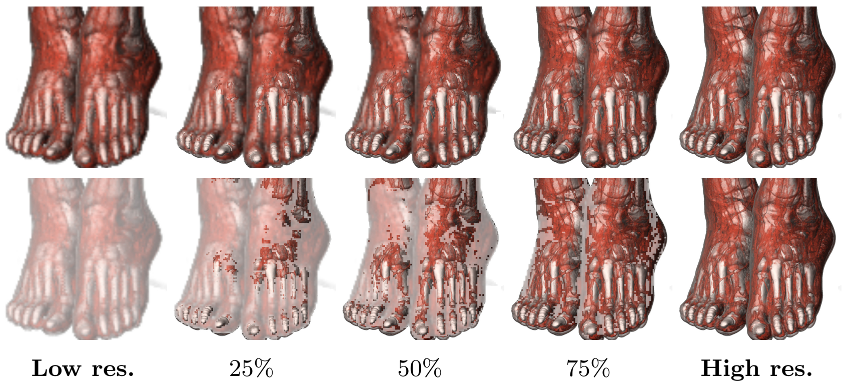


Figure 7.4: Vix dataset ($512^2 \times 250$ high res., $128^2 \times 63$ low res.). This sequence of images shows the transition process from the low-resolution to the high-resolution ray castings obtained by the *STiles* method. The top row shows the renderings as shown in the application, whereas in the bottom row, the low-resolution part of the same images is lightened in order to reveal the order in which the image is updated.

through the following steps:

1. **Low Resolution Ray Casting (during user interaction)**
    - The ray casting color is stored in a 2D texture T_{low}
    - The ray cost (number of ray samples) is stored in the alpha channel of T_{low}
2. **Tile Sorting (once after interaction finished)**
    - Once the user interaction stops, the screen space is split into tiles, and then, tiles are sorted by cost (using a series of compute shaders), generating two correlation maps that allow converting between unsorted and sorted tile coordinates.
3. **Progressive High Resolution Ray Casting**
    (a) The high-resolution ray casting of a few tiles (rendered in order) is performed until a fixed time budget is reached

    (b) The current image is composited, selecting either the high-resolution pixels from T_{high} when already computed, or the low-resolution ones from T_{low} otherwise

    (c) In the next frame, the process is resumed from (a) until all the tiles are rendered

During user interaction, a standard low-resolution ray casting for interactive rendering is performed in a fragment shader (see step 1 of Figure 7.3). At each pixel, together with the low-resolution color in the RGB channels, the number of ray samples is stored in the alpha channel of the output texture T_{low} as an estimation of the ray cost. This ray cost approximation is crucial for the main goal of the algorithm.

The second step starts once the user stops interacting. The viewport is then divided into small tiles, and these tiles are in turn sorted according to the ray cost hint provided by the previous stage (see step 2 of Figure 7.3), by means of a few compute shaders that implement a GPU radix sort algorithm [35]. The sorting pipeline proceeds in three steps, each one carried out by a compute shader: *i)* Group counting, *ii)* Group offset setting and *iii)* Sorting. During the first step *i)*, the tiles are grouped by cost, so that we finally have a counter of the number of tiles belonging to each group. We consider the cost of a tile to be the number of ray samples (previously stored in the alpha channel of T_{low}) at the center of the tile. The second stage *ii)* scans these counters to establish an offset for each group of tiles so that they can be later placed in an output texture without overlapping. Finally, the third and last step *iii)* proceeds by sorting tiles, placing them into the right position defined by their group offset, depending on their cost. The actual outputs of this compute shader are two texture maps that allow translating from sorted to unsorted tile coordinates and vice versa.

Finally, the last step of *STiles* corresponds to the progressive ray casting, carried out again by a fragment shader. It renders a variable number N of screen tiles, in order, in a separated 2D texture alias of sorted tiles. Thanks to the mapping textures produced in the sorting stage, the tiles can be rendered in strict order. The variable number of tiles depends on a fixed time budget (0.1 seconds in our case). The algorithm proceeds by rendering a window of N tiles. After rendering these N tiles, the elapsed time is measured in order to know if the time budget has been exceeded, and in this case, it does not render any more tiles during this frame. If the budget has not been exceeded, it renders N more tiles until the time budget is reached. The final image is composited by either selecting, for each pixel, the high-resolution ray casting color if available (again, using the mapping textures produced in the sorting stage to know its position in the sorted tiles texture), or the low-resolution ray casting color otherwise. This last step is repeated in successive frames, rendering as many tiles as possible without exceeding the fixed time budget, until the whole high-resolution ray casting image has completely substituted the low-resolution one (see step 3 of Figure 7.3). Figure 7.4-top shows the transition effect of this technique (in Figure 7.4-bottom, color is modified to better perceive the boundary between the low-resolution and the high-resolution pixels).

7.4.3 Discussion of decisions and alternatives

We have described two different strategies based on GPU ray casting for the incremental rendering of high-resolution volume datasets. Both are fast and complete the render of the final image quick enough to be considered good candidates for our purposes. One of the main strengths of the *FBSlabs* method is its low requirements regarding GPU specifications and OpenGL version. As our architecture is based on the use of 3D textures, a minimum version of OpenGL ES 3.0 is needed, but a different implementation that makes use of 2D textures to store the dataset in GPU memory could support lower versions of OpenGL.

In this sense, *STiles* has stricter requirements, demanding a minimum version of OpenGL ES 3.1 on mobile devices, due to the usage of compute shaders, which were not available in previous versions. For this reason, not only old graphics chips but also WebGL platforms, which today still do not provide a sufficiently updated version of OpenGL (and thus compute shaders are also not available), are not able to use *STiles*.

We have decided to implement the sorting step of *STiles* with a GPU radix sort [35] using compute shaders. This sorting strategy performs efficiently enough for our purposes, yielding negligible computation times, so the interactivity is not compromised. An implementation of this method in CUDA was also demonstrated to outperform other sorting algorithms in modern GPUs [73]. Another version based on fragment shaders could have been implemented with the aim of enabling older devices to execute *STiles* [38]. However, too many rendering passes are required to carry out the task (with a complexity of $O(n \log^2 n + \log n)$), yielding a serious penalty on mobile GPUs and likely providing less interactive results.

We have implemented some other alternatives for progressive ray casting with less satisfactory results. One of our first experiments was based on a naive separation of the high-resolution viewport into several tiles. We configured various splitting sizes: we found a grid of $8 \times 8 = 64$ tiles to be the optimal case for this technique, which was raising the completion time to at least one second due to the number of frames (64) needed to finish the render. Unfortunately, the transition between the low and high-resolution images was not pleasant due to its blocky appearance. This effect could be alleviated by increasing the number of tiles, but this would increase the total render time. Furthermore, we implemented and tested an early version of *STiles* that consisted in sorting individual rays instead of tiles, also using compute shaders. Although the idea of sorting seemed sound, the performance also dropped (see Section 7.5.1). Again, we believe that this is due to the fact that dealing with single rays breaks texture access coherence.

Another approach we implemented was a simple form of progressive ray casting (we name it *Simple* in what follows). It is a screen space refinement method that consists in starting with the render of a low-resolution ray casting image, and then progressively sampling new high-resolution rays on the screen surface at each frame until a time budget is expired, finishing when all the pixels of the high-resolution image have been computed. The high-resolution pixels computed at each frame are accumulated in an extra texture so that they can be reused in subsequent frames. The sampling scheme for the selection of new high-resolution pixels (to generate new rays) at each frame, is analogous to other techniques of progressive ray casting mentioned in Chapter 3.5. The new rays are generated and computed at each frame, covering the screen in a uniform way over time. We have tested two slightly different approaches, one where the pixels for the high-resolution ray sampling are selected randomly (referred to as *Simple random* in the figures), and another one where the pixels are selected using a uniform distribution over the screen surface (labeled as *Simple structured*). The number of refinement steps is variable and depends on the number of rays computed at each frame without exceeding the fixed time budget, which is directly related to the complexity of the rendering process (i.e., resolution of the model, opacity of the Transfer Function, viewport resolution, etc.). The transition effect between the low-resolution and the high-resolution images was highly smooth, up to the point of almost not noticing the transition. We used the same performance optimizations used in the other methods presented (i.e., ERT and ESS). However, the performance of this approach was worse than our proposed methods (see Section 7.5.1). We hypothesize that the pseudo-random distribution of rays was preventing all kinds of cache usage on the GPU, thus increasing the render time at each frame and achieving a much less interactive experience. We present an evaluation of this method in Section 7.5 together with the evaluation of our proposed techniques.

7.5 Evaluation and results

Rendering high-quality images of a relatively large dataset on low-performance devices such as mobile devices is a task that requires a significant amount of time. We have proposed an incremental approach that splits this process into separated steps that are completed over subsequent frames. This way, each step can be executed during an application frame not exceeding an acceptable amount of time. This avoids blocking the application and provides smooth interactivity, allowing the interruption of the high-quality render at any time if the user desires to continue interacting. Our two proposed methods accomplish this task quickly and in a visually pleasing way. So, from the point of view of the user, the only visible difference is the transition from the low-quality image to the high-quality image.

We performed several experiments to measure the advantages of both approaches in terms of performance (Section 7.5.1) and visual quality (Section 7.5.2). The experiments were run on two mobile devices, a Motorola Nexus 6 (equipped with an Adreno 420 GPU and a screen resolution of 1440×2560) and an Huawei Nexus 6P (equipped with an Adreno 430 GPU and a screen resolution of 1440×2560). On both devices, the viewport resolutions were scaled to half the screen size on both axes for the high-resolution ray castings (720×1280, which is still a good resolution due to the small pixel size given on these devices' screens) and to one eight of the screen size for the low-resolution ray castings (180×320). We used datasets of different resolutions with Transfer Functions having different levels of transparency: Vix ($515^2 \times 256$), Head ($512^2 \times 485$), Obelix ($256^2 \times 780$), Chamaleon (512^3) and Melanix ($256^2 \times 602$).

7.5.1 Transition from low to high resolution: performance

FBSlabs distributes the workload over time by splitting the rays into segments. At each frame of this progressive method, a limited number of ray casting samples is fixed, so the maximum number of samples within the ray casting shader, for a single frame, is $O(V_w \times V_h \times N)$, where $V_w \times V_h$ is the total number of pixels in the viewport and N is the fixed number of samples to take from each ray segment during a single frame of the incremental render. We have fixed $N = 40$ in our experiments so that a small loop is performed for each pixel in the viewport at each frame. Besides the rendering of each slab, the amount of time required for blending both, the low-resolution and the high-resolution images, is negligible. Some results are shown in Figure 7.5 (*FBSlabs* series). On average, our experiments obtain completion times under 1 second for the tested models.

In order to improve *FBSlabs*, we made a test that stored per-ray accumulated opacity after each frame, so that a global ERT is enabled. However, this implementation requires an extra pass to copy the high-resolution results into another texture that can be queried during the next frame to know whether or not the current ray/pixel was completed and can be discarded. Unfortunately, this extra pass incurs a time penalty that is larger than the benefits obtained from ERT. The algorithm can still perform per-slab early ray termination, but it will not avoid starting the ray traversal for the next slab in the next frame.

In *STiles*, the workload is split into screen-space tiles that can have different costs depending on the length of the rays they contain. Then, they are sorted before proceeding to the progressive ray casting step. The sorting step cost is actually negligible, and it is computed only once after each user interaction (see step 2 of Figure 7.3). We base our strategy on the experimental results shown

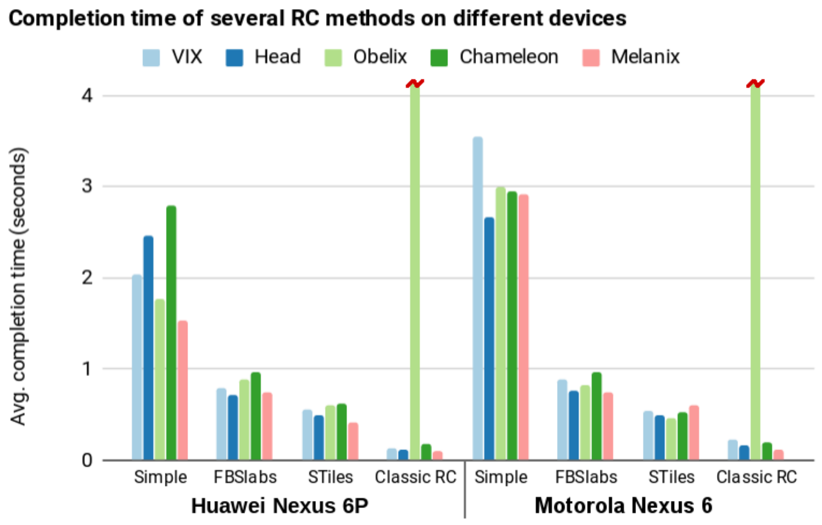
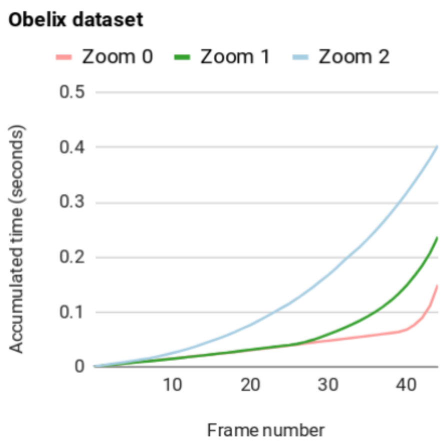


Figure 7.5: Average completion time (in seconds) of progressive ray casting methods run on different devices for several datasets. These times are an average measure calculated by performing the transition process from 3 different zoom levels with different screen pixel coverages, and 20 different camera positions uniformly distributed on the surface of a surrounding sphere for each zoom level (60 camera configurations in total). Note that the measurements for the *Classic RC* complete faster in average, which is the expected behavior as the rendering task is not split over several frames. However, this does not implies better interactivity than progressive methods, because these distribute the workload over several frames, returning the control to the application more frequently. Note also the high bars in the *Classic RC*, indicating that some renderings were not completed due to an application crash.

in Figure 7.6. If tiles are sorted by increasing cost, it can be observed that the accumulated time of the incremental rendering along subsequent frames (when a fixed number of tiles is rendered at each frame) increases in a non-linear way, due to the obvious fact that rendering the first tiles is faster than rendering the last ones. Our strategy, based on the charts in Figure 7.6, is to render more tiles in the first frames and a lower number of tiles in the last frames to compensate for their higher cost. Based on the shape of the curves in these charts, we estimate a tile budget for each subsequent frame that guarantees an estimated time budget of 0.1 seconds per frame. Estimated tile budgets are decreasing from the first frame to the last one, resulting in a greater number of tiles being rendered in the first frames and on a stable frame rendering time. As shown in Figure 7.5, we achieve completion times faster than *FBSlabs* method (approximately half the time for all the tested datasets on all devices).



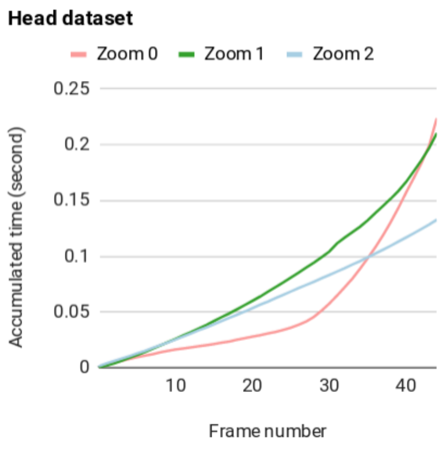


Figure 7.6: Charts showing the accumulated time over subsequent frames in STiles for two datasets. The lines correspond to different levels of camera zoom, corresponding Zoom 0 to the smallest, and Zoom 2 to the largest screen pixel coverage. A fixed number of tiles is rendered at each frame, and the tiles have been previously sorted by increasing cost. Last frames obviously take longer to finish.

One could argue that rendering tiles in ascending order in *STiles* implies rendering big empty regions of the screen first (which should have cost zero) whenever the footprint of the proxy geometry is much smaller than the actual screen resolution. The ideal procedure would be to directly discard those tiles without effective work to process, or those not overlapping the proxy geometry. However, discarding tiles with zero cost is not reliable, as tile costs are computed from a low-resolution image rendering, so the ray costs in low resolution might be inconsistent with the same rays traversing the high-resolution dataset. In addition, we actually classify each tile by the cost queried from a single sample position at its center, which is a fast approximation that discards many texels with information (a sample fetched from the center of the tile could have cost zero, but one of its corners could have a high costs, if the region covered by that tile corresponded to an edge of the model). However, this issue is not a problem, as the rendering of empty, and almost empty tiles, completes instantly when the fragment shader discards rays not intersecting the proxy geometry, so it is actually normal completing all the empty regions and part of the effective ray casting workload during the first frame.

As previously commented, we tested an initial version of *STiles* that consisted in sorting individual rays, achieving poorer performance. We were then inspired by an analysis of the rasterization patterns followed by several GPUs

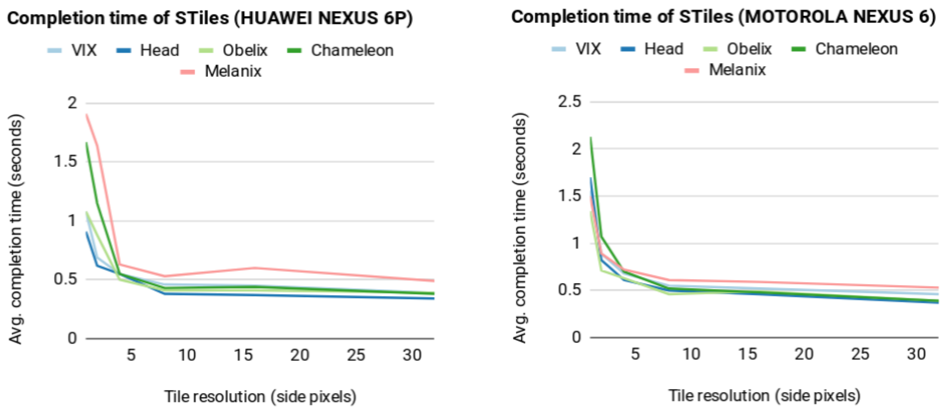


Figure 7.7: These charts show the overall completion times (in seconds) obtained for the *STiles* algorithm under several tile size configurations. The tests were run on two different devices (Huawei Nexus 6P on the top, Motorola Nexus 6 at the bottom) with several datasets. The tested tile sizes were: 1^2, 2^2, 4^2, 8^2, 16^2 and 32^2. We can see how the completion time decreases as the tile size increases. More precisely, the performance gain is particularly low for sizes greater than 8^2, which is actually the size of the rasterization patterns used by those GPUs.

in [37], where the authors were able to reveal the order in which pixels are rendered by the GPUs by means of using atomic counters in a fragment shader. Based on this observation we performed an analysis of the performance by running some tests, packing groups of rays in tiles of several sizes (see Figure 7.7). As expected, increasing the tile size boosts performance. The rationale behind this is that packing neighbouring rays together takes advantage of the 3D texture cache. Following this argumentation, performing the whole render at once would achieve an optimal result. However, the measurements shown in Figure 7.7 are averaged over a big variety of camera configurations where some renderings are generated very quickly, and others can take much longer (e.g., the Body model seen from above through its longest axis), and they could provoke the aforementioned application crash issue if not split over time. We finally decided to use a minimum tile size of 8×8 pixels, as the performance gain considerably decreases for larger tile sizes. As shown in Section 7.5.2, this tile size achieves a good compromise between the rendering time and the perceived transition between different frames.

We also tested the performance of the *Simple* progressive ray caster. The achieved completion times were the higher among all methods (see Figure 7.5). This is due to the distribution pattern followed to generate rays for the high-resolution ray casting. It does not take into consideration any locality pattern, breaking the spatial coherence and not making possible the use of the 3D texture cache, finally increasing the total completion time. In addition, we executed performance tests of a classic non-progressive ray casting algorithm to compare the achieved times with the result of our proposed progressive methods. The average rendering times obtained may seem lower than our two proposals (see Figure 7.5, *Classic RC*). However, these are averaged numbers only from successful frames. Other images, taking longer to be rendered stall the application until finishing, not giving the user the opportunity to interact. Some others cannot even be averaged as they make the application crash due to long stalls (this is the case of the *Obelix* dataset when visualized along its longest axis, as the used Transfer Function is barely opaque, and that generates very long rays). Furthermore, it is desirable to receive partial results of the final image right after finishing interacting (even if it takes a bit longer to complete the image), which gives the user a hint to perceive that the application is actually working. This performance is again not offered by classic non-progressive ray casting algorithms.

Some extra tests were performed in order to measure and compare the interactivity of the presented progressive ray casting methods. As seen in Table 7.1, all progressive methods present an acceptable frame rate in all cases during the generation of the high-resolution image, being *FBSlabs* the more interactive, followed by *STiles*, and being the *Simple* progressive method in third place. Note, however, that the classic non-progressive ray casting provides worse frame rates and hence bad interactivity in average, provoking application crashes occasionally, as shown in Figure 7.5.

7.5.2 Transition from low to high resolution: visual effect

The visual effect of the transition between low-resolution and high-resolution images obtained by *FBSlabs* and *STiles* is quite different. Figures 7.2, 7.4, 7.12, 7.13, 7.14 and 7.15 (at the end of the chapter) show the progression of each method during the transition time with renderings of several datasets, visualized with Transfer Functions designed with different colors and opacities.

The progressive *FBSlabs* method has the effect of the high-resolution image appearing on top of the low-resolution one (see Figure 7.8, *FBSlabs*) and completes gradually replacing the low-resolution image in front-to-back order. During the incremental rendering, the final color that is presented on the screen is the composition of the high-resolution image on top of the remaining part of

Nexus 6

	Simple			FBSlabs			STiles			Classic RC		
Dataset	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Vix	8.036	11.612	8.845	10.752	32.641	20.625	12.510	23.077	16.525	2.243	11.207	4.979
Head	8.166	11.188	9.731	17.599	37.492	25.424	11.072	23.272	16.208	3.309	11.413	6.444
Obelix	7.857	10.293	8.838	13.866	40.319	23.907	9.291	26.863	17.068	×	13.378	6.714
Chamaleon	7.905	10.836	9.758	17.135	37.281	25.357	13.511	23.705	16.898	2.045	6.9417	4.481
Melanix	7.928	10.291	8.654	15.196	42.841	25.553	13.058	28.441	19.734	4.160	19.548	9.981

Nexus 6P

	Simple			FBSlabs			STiles			Classic RC		
Dataset	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Vix	9.042	10.811	9.658	18.329	47.705	28.866	13.419	24.141	16.621	3.841	16.785	7.963
Head	7.805	10.519	9.653	23.056	51.682	31.548	13.786	24.439	17.263	4.039	14.899	8.704
Obelix	8.650	12.541	9.683	17.713	48.328	26.762	9.684	26.289	16.442	×	19.372	9.076
Chamaleon	8.824	12.948	9.597	17.136	47.975	26.198	12.117	21.777	15.770	2.343	11.542	6.379
Melanix	9.188	13.379	10.130	18.545	54.088	26.918	13.173	32.586	20.303	4.998	22.587	12.697

Table 7.1: These frame rates reflect the interactivity of the presented progressive ray casting methods with respect to a classic non-progressive ray casting algorithm on two different mobile devices. All progressive methods perform interactively in all cases during the generation of the high-resolution image, being *FBSlabs* the more interactive, followed by *STiles* and being the *Simple* progressive method in third place. Note, however, that the classic non-progressive ray casting provides worse frame rates and hence bad interactivity in average, provoking occasional application crashes as shown in Figure 7.5.

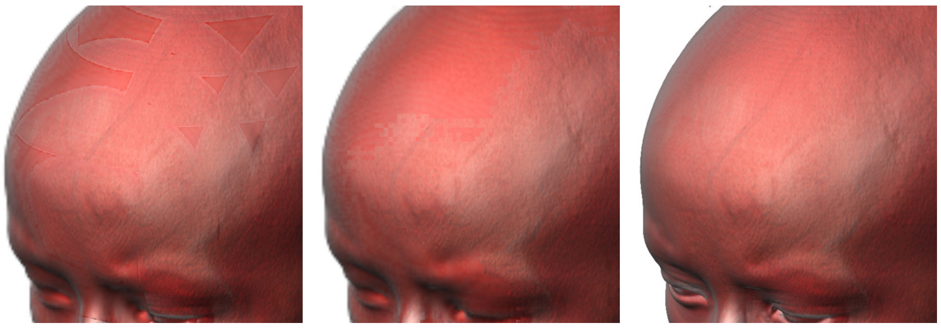


Figure 7.8: Detail of an intermediate step during the high-resolution transition process (Head dataset $512^2 \times 485$ high res., $128^2 \times 122$ low res.). In *FBSlabs*, the transition boundary is more evident and reveals patterns generated by the fact that ray sampling proceeds front-to-back from the proxy geometry. The boundary is less perceivable in *STiles*, which furthermore has a pseudo-random transition pattern that makes it less evident over time.

the low-resolution image using alpha blending. An issue regarding this way of compositing images is that we are mixing viewport resolutions. In the context of ray casting, this means two things. The first one is the fact that the rays in the low-resolution image do not perfectly match rays in the high-resolution image. And the second one is that we are performing an upsampling of the low-resolution image, so we are interpolating color to match the sizes of both images. This sometimes results in slight seam artifacts revealed in the boundary between the high-resolution and the low-resolution models.

STiles also reveals the final high-quality image gradually, but in this case, small tiles with the corresponding part of the high-resolution image appear in a pseudo-random order (see Figure 7.8, *STiles*). It also gives the impression of completing the result in some sort of front-to-back order (or back-to-front order, it actually depends on the sorting strategy) but each tile with high-resolution color that has been computed completely replaces the initial low-resolution color, instead of compositing the high-resolution color over the low-resolution color as in *FBSlabs*. We can choose between sorting tiles in increasing or decreasing order of ray cost. In the first case, tiles with small cost (e.g., those with rays that become completely opaque very quickly) are rendered first. This way, models visualized with Transfer Functions designed to reveal opaque isosurfaces exhibit a transition effect that gives the perception of most parts of the final image appearing first, and then the silhouettes appearing in the end. A reverse sorting strategy, starting from tiles with an estimated high cost

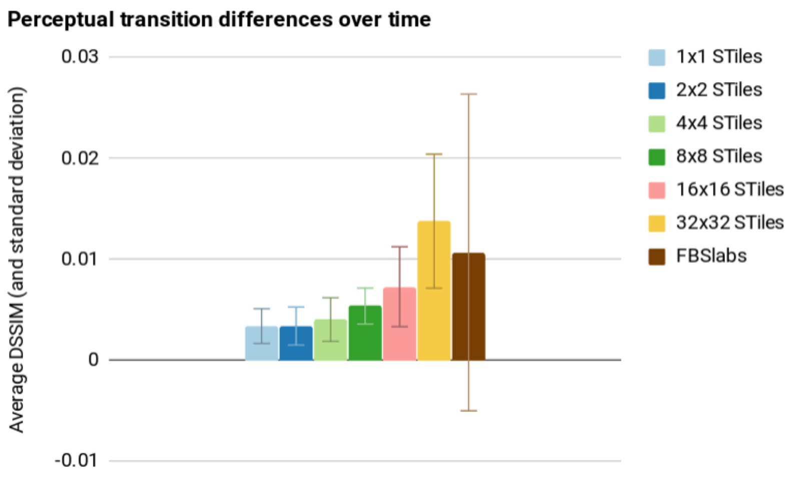
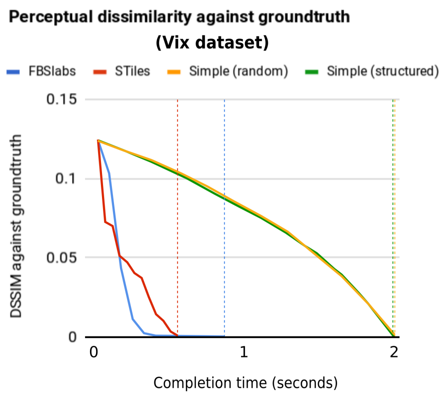


Figure 7.9: This chart shows the measured average perceptual error (and its standard deviation) on the transition process (going from the low-resolution image to the high-resolution image). The perceptual metric used is the structural dissimilarity metric (DSSIM). The average error was computed using pairs of consecutive frames in several series of the incremental ray casting process. We can observe that the transition becomes perceptually more evident (i.e., has a higher error measure) as the tile size increases, being significantly greater for tile sizes greater than 8^2 (note that 16^2 has a considerably higher standard deviation).

and then rendering tiles in decreasing order gives the contrary visual effect: first, translucent areas and most silhouettes are revealed, and then opaque areas with a little translucent component are computed in the last place. We decided to sort tiles by increasing order because, in most cases, the effect it achieves is more desirable, and furthermore, the transition achieved gives the perception of completing sooner due to the fact of rendering more tiles in the first frames.

As explained in Section 7.5.1, we empirically determined a lower bound of the tile size (in pixels) based on an analysis of the GPU rasterization pattern [37] and a series of experiments regarding performance (Figure 7.7). These experiments show a tendency to gain performance when increasing the tile size. However, *STiles* performs a tile-based rendering, and it consequently presents a blocky transition effect that becomes more evident when the tiles are too large. To determine an appropriate tile size, we also performed a series of experiments to measure the transition changes over time using a perceptual structural dissimilarity metric (DSSIM) [85]. Figure 7.9 shows averaged perceptual differences over time. The perceptual differences shown in the chart are



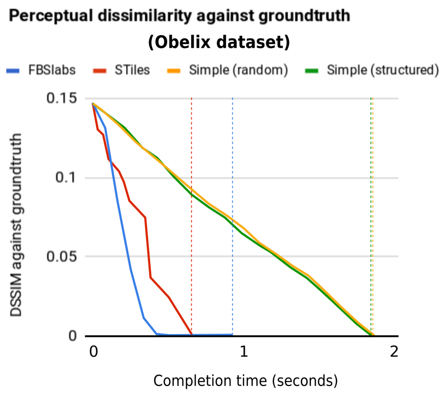


Figure 7.10: Perceptual changes of progressive methods over time. The data was taken from rendering the *Vix* and the *Obelix* datasets (see Figures 7.4 and 7.14) and comparing each frame resulting image with the final image (ground truth). Here we can see that *FBSlabs* is the fastest in converging towards the final image, even when the Transfer Function is more transparent, such as in the case of the *Obelix* dataset. *STiles* and the *Simple* progressive ray casters converge to the final image more smoothly, being *STiles* the most performant approach.

obtained by comparing each intermediate frame with the previous one. Based on the obtained results, we decided to fix the tile size to 8×8, as the perceptual differences increase for larger tile sizes. This size is actually the lower bound determined in the previous section, and also the size of the tiles generated by the rasterization process on these GPUs. This size is small enough so that the blocky nature of this method is not evident or annoying during the transition between the low-resolution and the high-resolution images.

We did another set of tests to measure and evaluate the quality of the transition of both methods on several datasets, also using perceptual metrics (DSSIM). Figure 7.10 shows the perceptual transition profile of our presented progressive methods (*FBSlabs* and *STiles*) and of the two different approaches of the *Simple* progressive ray caster, one distributing rays in a pseudo-random order (random), and another in a more structured way (structured). These tests were done using the *Vix* and the *Obelix* datasets (see Figures 7.4 and 7.14), which are visualized using Transfer Functions with different levels of transparency. The charts show the perceptual image variation of each frame with respect to the final (ground truth) image. In both figures, we can see how *FBSlabs* converges to the resulting image faster than the other methods, approximating to the final result in the first frames. However, it keeps on executing during several frames until all the model has been rendered. This

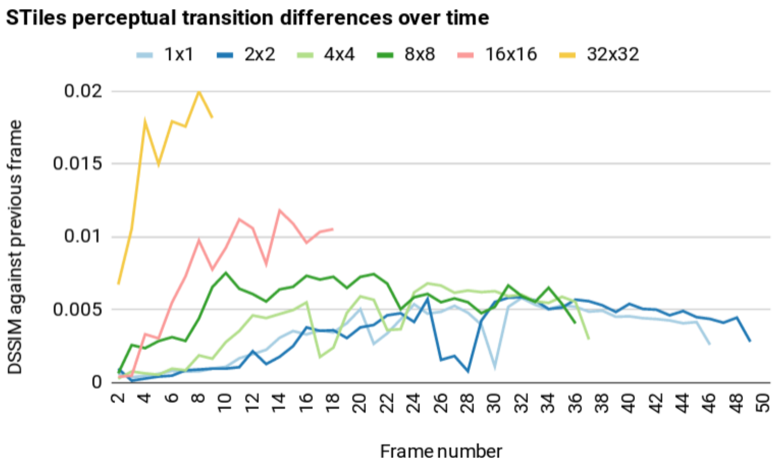


Figure 7.11: Perceptual changes of *STiles* over time using different tile sizes. Data taken from rendering the *Vix* dataset (see Figures 7.2 and 7.4) and comparing each pair of subsequent frames in the timeline. Larger tile sizes achieve higher perceptual changes between subsequent frames. This is actually normal considering that the final image completion is usually achieved in less frames when using larger tile sizes. Note that tiles of size 8×8 and smaller achieve similar measures over time, yet tiles of size 8×8 take less frames to finish among those *small* tile sizes (and less time, see Figures 7.7 and 7.5).

quick convergence is due to the front-to-back nature of the method, as the front part of the model usually covers most part of the image, yet the back part of it has less impact on the final result. The other methods have a more constant transition effect, being *STiles* better than the two approaches of the *Simple* method (which behave very similarly), as it converges to the final result uniformly in less time. These observations confirm the perception we had when analyzing the running application and our preference towards *STiles*, as it converges quite fast to the final image and keeps a gradual and smooth transition over time.

Figure 7.11 shows DSSIM measurements of each frame of the progressive render with respect to its previous frame for different tile size configurations in *STiles*. In this case, the charts show that the biggest tile sizes achieve a higher error, meaning that the transition is less smooth and more perceivable. However, tiles of size 8×8 and smaller have a similar profile. Taking this into account and considering the performance results in the previous section (see Figure 7.7), we decided to use tiles of size 8×8 as the default option.

Feature	**FBSlabs**	**STiles**
OpenGL version	Requires OpenGL ES 3.0 or lower if the 3D volume is managed with 2D textures.	Requires OpenGL ES 3.1 because it needs compute shaders.
Transition effect	High-resolution image appearing front to back. Major changes occur during the first frames. More perceivable seams between low-resolution and high-resolution models.	Better DSSIM perceptual results. Transition occurs more regularly distributed over time. Pseudo-random substitution pattern of the low-res image by the high-res one.
Transition time	Good average completion times. A small number of ray casting samples is fixed at each frame. High interactivity rate.	Better average completion times. A time budget is fixed for each frame that cannot be exceeded. At each step, as many tiles as possible are rendered. Good interactivity rate.

Table 7.2: Characteristic features of the *FBSlabs* and *STiles* methods for progressive ray casting.

7.5.3 Final remarks

Both *FBSlabs* and *STiles* are usable when generating progressive renderings of volume data. The presented performance tests show that they enable less powerful devices to render big volumes of data otherwise not feasible. Table 7.2 summarizes the main features of the two proposed algorithms. We recommend using *STiles* over *FBSlabs* whenever possible. It fits devices with OpenGL ES 3.1 (needed for the compute shaders). The results obtained for *STiles* are better both in performance and in visual quality as demonstrated in the previous sections. It completes the high-quality image in less time than *FBSlabs* and the perceptual variation over time as the transition advances is smaller, a fact that matches our visual assessment. Not far from it, however, *FBSlabs* is a good candidate to use in less powerful devices that do not provide compute shaders. Furthermore, even when running on more capable hardware, *FBSlabs* is a good choice on platforms such as WebGL, whose standard still does not support modern features such as compute shaders. Moreover, it could even be adapted for older devices that do not provide 3D textures using a scheme based on flat 3D textures or stacked 2D textures, for instance.

7.6 Conclusions and future work

In this chapter, we have proposed a multiresolution architecture based on ray casting aimed at achieving the interactive rendering of volume ray casting in less powerful devices, such as mobile phones and PCs with low-end and old graphics chips. We use a low-resolution dataset to perform interactive visualizations during user interaction, and the higher resolution version of the same dataset (that still fits the target's GPU memory) to perform a high-quality visualization each time the user stops interacting.

Our main contributions are two scalable methods for the progressive ray casting of high-resolution datasets that are able to decouple the rendering process into separated batches that can be rendered over subsequent frames: *FBSlabs* and *STiles*. These algorithms are able to provide an interactive user experience without application stalls at any time. Based on the performed experiments, we conclude that *STiles* achieves better results in both performance and visual quality than *FBSlabs*, as presented in Section 7.5. *FBSlabs* is, however, a good candidate for less up to date devices that do not provide modern GPU features (e.g., compute shaders).

In order to improve the quality of the visualization of the low-resolution datasets, we make use of some of the contributions presented in the previous chapters in this thesis. First, the coarse versions of the original dataset were generated using the feature-preserving downsampling filter presented in Chapter 4. Furthermore, the computation of colors and opacities in the coarser datasets were done using our algorithm, Adaptive Transfer Functions, explained in Chapter 6. With the use of these two techniques, the renderings obtained from coarse models of the multiresolution hierarchy preserve fine and small details that usually disappear when using conventional downsampling methods, and the perceived colors and opacities better match the colors and opacities of renderings of the original dataset. Besides the contributions in the previous chapters of this thesis, other standard methods to improve efficiency (ESS and ERT) and image quality (pre-integrated volume rendering) were incorporated in the algorithms presented in this chapter.

We have not included the use of pre-computed gradients (see Chapter 5) in the system here presented. Although it makes sense in terms of performance (fast retrieval of gradient data instead of several texture lookups to perform the computation on-the-fly) and in terms of quality (previous tests in desktop machines have demonstrated to improve this aspect), the fact that graphics memory is a scarce resource in mobile devices prevented us from using it. Right now, the low-resolution and the high-resolution datasets are stored in graphics memory simultaneously. The low-resolution dataset is chosen so that the frame rate obtained achieves a high degree of interactivity. The high-resolution dataset, however, is the highest resolution dataset from the multiresolution hierarchy that fits the device memory. Besides that, using an extra volume with pre-computed gradients and having it in the GPU is not possible because there is not enough memory available. Then, the only option in order to use pre-computed gradients would be using lower-resolution datasets, but that would decrease the overall quality of renderings anyway. Out-of-core techniques could be employed in order to incorporate datasets of higher resolutions, and with more per-voxel information such as pre-computed gradients, or alleviating the simultaneous usage of graphics memory so that the low-resolution dataset could use fast, high-quality pre-computed gradients. This is however out of the scope of this thesis.

Regarding *STiles* a slight improvement would be the ability to split the current individual ray batches into several parts. It is not likely that our algorithms are going to deal with volume datasets large enough to make the device stall by only rendering a single ray group. However, that could happen if rays were long enough, which could be solved by also allowing incremental rendering of individual tiles.

Current sizes of really large datasets ($\geq 1024^3$) cannot fit current GPUs' memory specifications. A possible way to extend our architecture is the implementation of an out-of-core block based scheme that allows fetching blocks as needed during the high-resolution rendering process, so our progressive rendering algorithm could require the needed blocks from the storage memory or server at each frame. At first sight, it seems that the implementation of a block-based on-demand architecture like this could be easier to extend *FBSlabs*, which already performs an object space partition to carry out the progressive render, rather than *STiles*, which is a screen space approach.

7.7 Publications

The techniques presented in this chapter have generated the following paper:

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (2018) *Progressive Ray Casting for Volumetric Models on Mobile Devices. Computers & Graphics. DOI: https://doi.org/10.1016/j.cag.2018.02.007* [19]

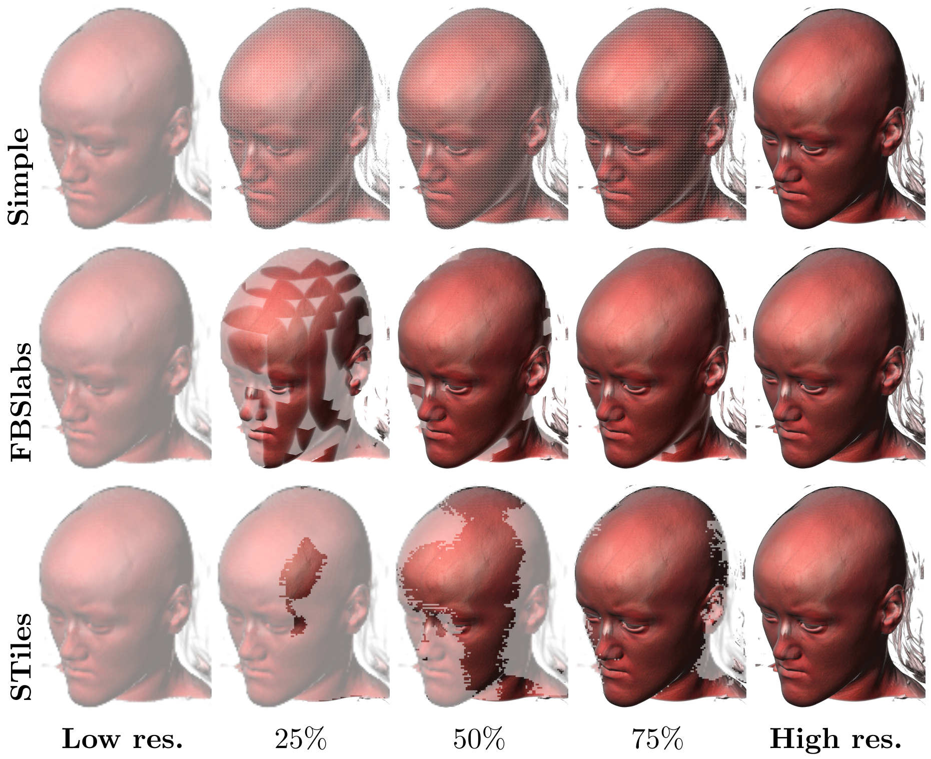


Figure 7.12: Illustration of the transition in the presented algorithms for the Head dataset ($512^2 \times 485$ high res., $128^2 \times 122$ low res.). These figures do not correspond to the actual rendering, but we modified them in order to show which parts of the image are updated over subsequent frames in both algorithms: the region that has not yet been updated with the high-quality rendering is shown with a semi-transparent look. Note that *Simple* has the more incremental transition. Note also that *FBSlabs* has homogeneous boundaries that are easier to perceive during the progression than *STiles*, and *STiles* provides a pseudo-random transition pattern that is more difficult to notice during the incremental rendering (see Figure 7.8).

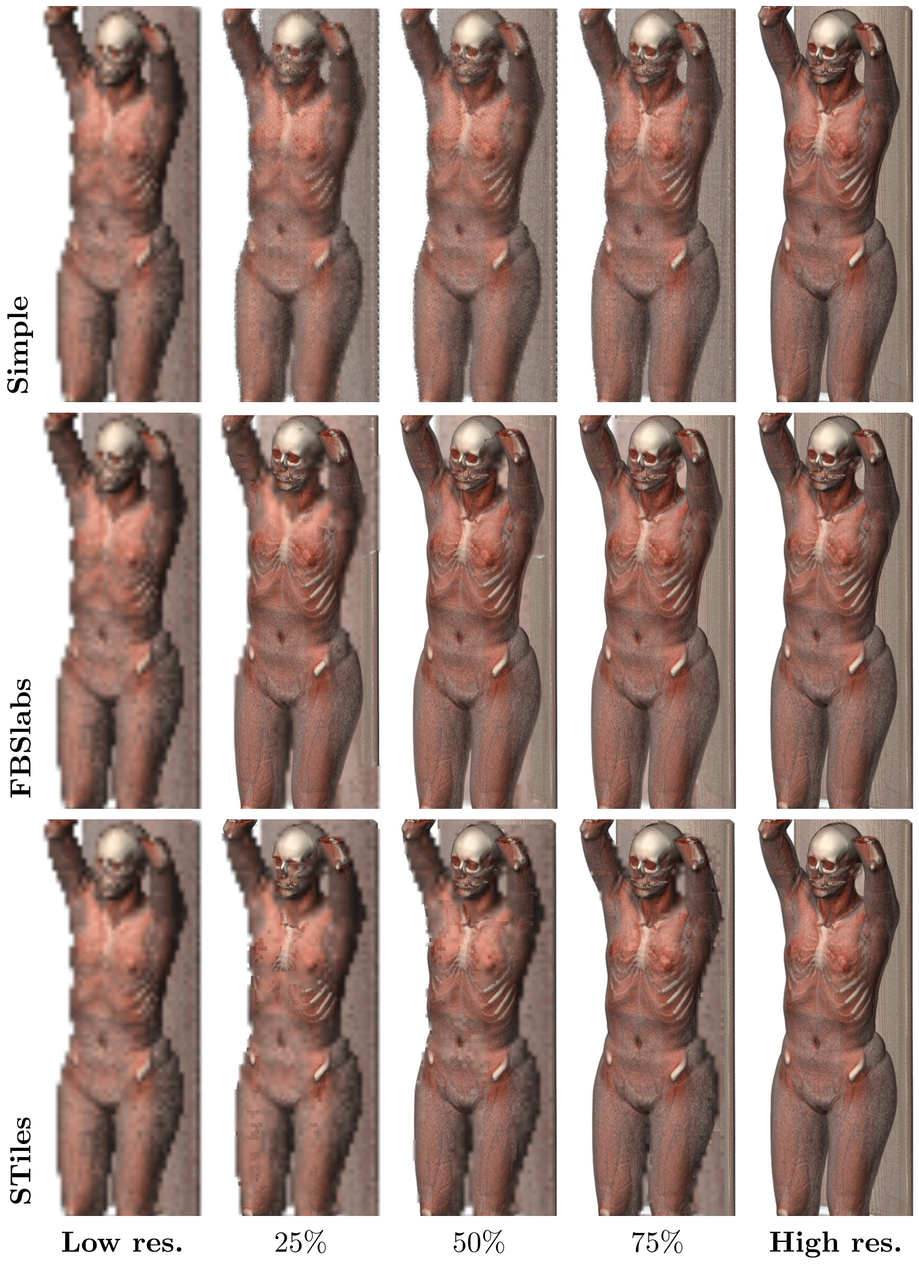


Figure 7.13: Melanix dataset ($256^2 \times 602$ high res., $64^2 \times 151$ low res.). Transition effect of the two proposed incremental ray casting algorithms using a Transfer Function with almost opaque colors.

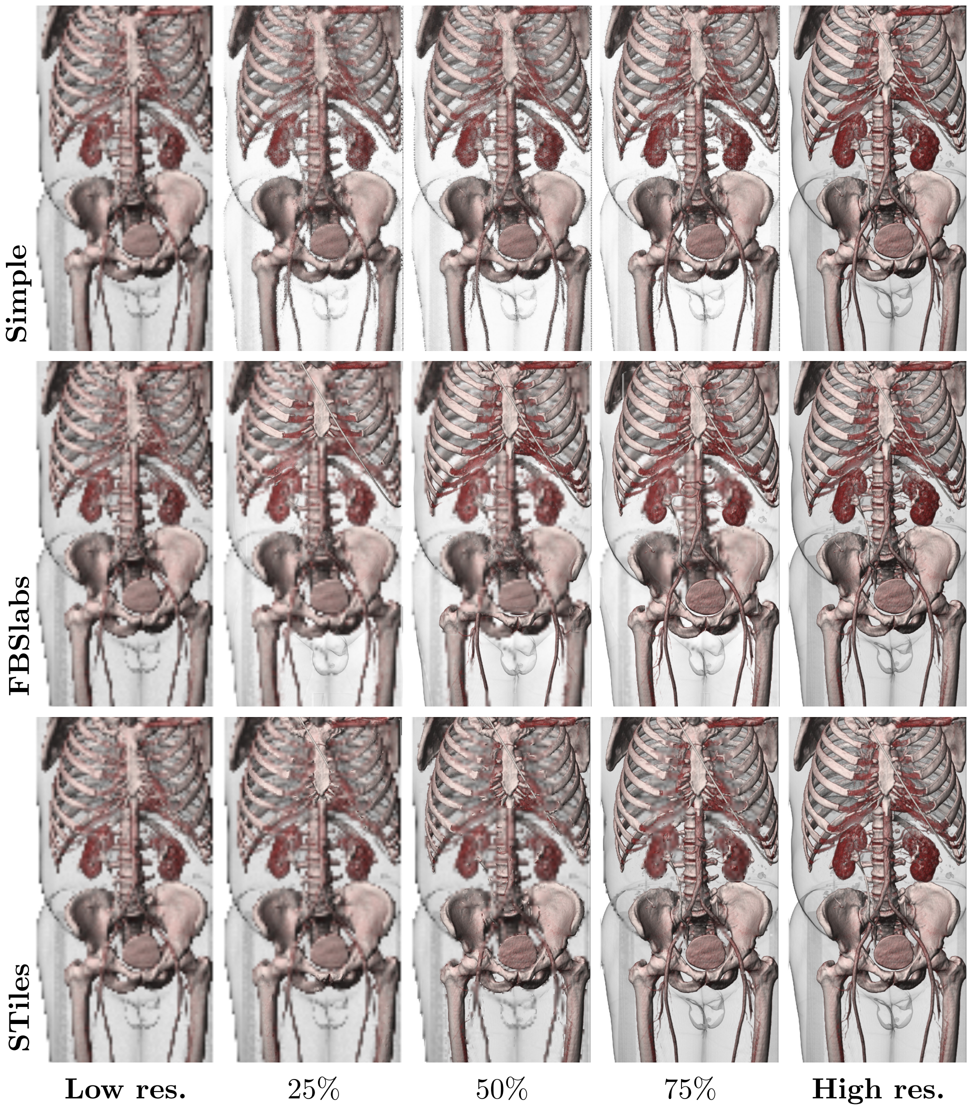


Figure 7.14: Obelix dataset ($256^2 \times 780$ high res., $64^2 \times 195$ low res.). Transition effect of the *Simple*, *FBSlabs* and *STiles* incremental ray casting algorithms using a Transfer Function with some opaque colors (bones, kidneys, etc.) and semitransparent colors (skin).

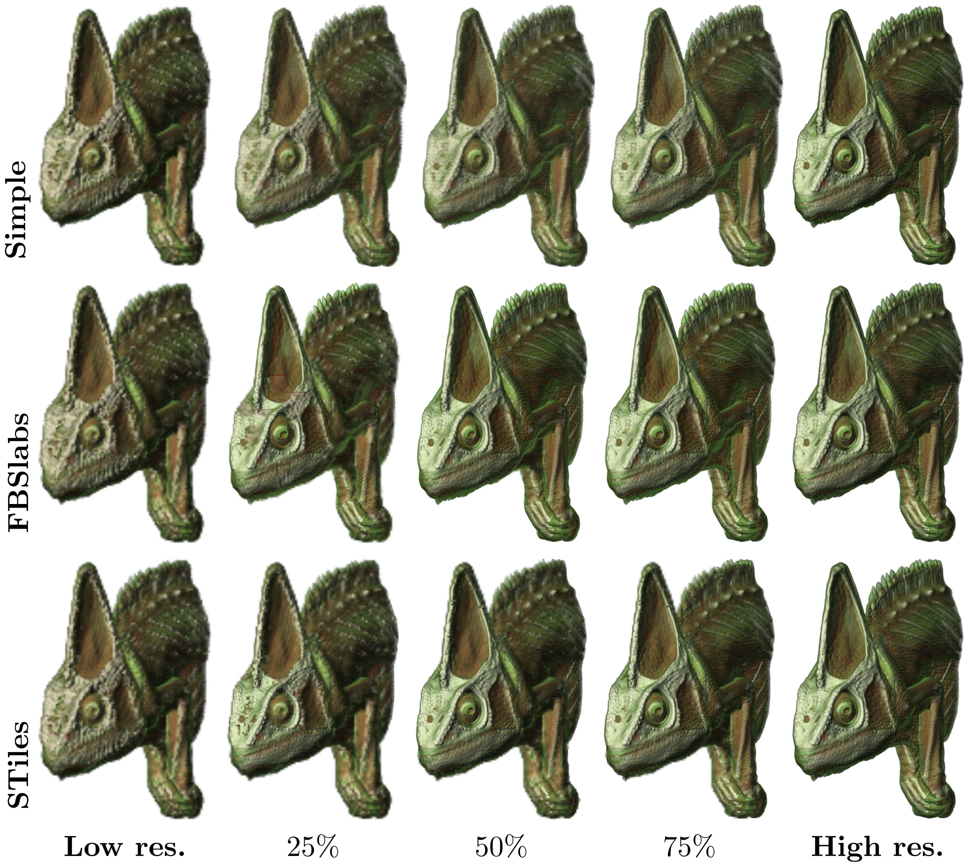


Figure 7.15: Chameleon dataset (512^3 high res., 128^3 low res.). Transition effect of the two proposed incremental ray casting algorithms using a Transfer Function with some opaque colors (bones, muscles, etc.) and semitransparent colors (skin). Although we are mainly focusing on medical datasets, the presented algorithms are perfectly suited for any other kinds of volume datasets such as this one.

8 Conclusions and Future Work

This chapter ends this document by drawing some conclusions out of our research work and pointing out the main contributions achieved during the time of this thesis. Also, a few lines of future work that would complete some aspects of the presented material or would be worth exploring are mentioned. Finally, a list of the published papers that have been the results of this thesis is given at the end of the chapter.

8.1 Conclusions

During this thesis, we have addressed the problems arisen from visualizing medium to large models (512^3 voxels) in low-end or mobile devices, whose capabilities are far below their desktop counterparts. The requirements of the developed methods have been the generation of good quality results (i.e., as similar as possible as possible to the original data) striving to avoid significant performance penalties, thus obtaining small pre-processing times (when pre-processing is necessary at all), and achieving high frame rates that ensure interactive visualizations.

We have approached the previously mentioned problems by embedding specific techniques in the visualization pipeline (see Section 2.1) both in the data processing stage (e.g., filtering and generation of multiresolution hierarchies) and in the data visualization stage (GPU-based ray casting).

The main contributions in the course of this thesis have been:

- An analysis of the performance of various downsampling filters for volumetric models.
- A novel Gaussian-based fast and feature-preserving downsampling filter to generate multiresolution volumetric models.
- A downsampling filter for pre-computed gradients to better preserve gradient directions, and an effective method for their encoding and storage.
- An efficient algorithm that interactively generates Transfer Functions specially tailored for coarse resolution datasets to improve the quality of renderings.
- Two novel techniques for the incremental high-quality rendering of big volume datasets in commodity hardware and mobile phones that ensure interactivity.

To properly improve downsampling methods, we have started the thesis with an evaluation and comparison between several existing downsampling filters for scalar data. Based on these observations, we have developed two downsampling techniques, one for scalar fields and another one for gradient directions.

The Gaussian-based feature-preserving downsampling filter for scalar data we have designed is able to generate coarser resolution models, obtaining smooth results and preserving small and fine features that usually disappear with standard downsampling methods such as averaging or applying a Gaussian filter before subsampling. Besides achieving good results, the filter performs fast and is completely automatic. We have shown several comparisons between the results obtained with our proposed filter and several other common approaches used for the downsampling of volume models.

Concerning gradients, we have developed a technique to perform filtered downsampling of pre-computed gradient data that is able to preserve gradient directions better than computing on-the-fly gradients directly from the downsampled scalar fields. The filter we have proposed avoids common artifacts that appear during downsampling gradient data due to regions where gradients are not well defined. Moreover, we have presented an effective method to encode and store gradients that maximizes the number of representable directions using a representation of three 8-bit components.

Regarding Transfer Functions, we have presented a method that computes Transfer Functions specially tailored for coarse resolution datasets so that the quality of renderings is highly improved with respect to using the originally designed Transfer Function. The technique requires a light pre-process to construct downsampling 2D joint histograms for each coarse dataset in the multiresolution pyramid, and performs interactively generating adapted Transfer Functions whenever the original Transfer Function is modified by the user.

Finally, we have designed a multiresolution, scalable rendering framework that uses a low-resolution configuration (dataset and viewport) during interactive manipulations of the viewpoint, and a high-resolution one intended for static inspection/observation when the interaction stops. In order to make the transition from the low-resolution and the high-resolution images, we have presented two progressive ray casting strategies that ensure interactivity during the process of rendering high-quality images in mobile devices, allowing user interactions at any time.

8.2 Future work

There are still pending tasks for further exploration and ideas that we would like to test. Regarding the feature-preserving filter, it would be interesting to explore the possibility of generating adaptive kernel sizes depending on the statistical information derived from each point in the dataset. Our idea is that smoothing regions with little variance and doing the opposite for regions with higher variance would contribute achieving smoother results while still preserving sharp, fine-grained features.

Adaptive Transfer Functions perform an adaption of the color based on a global analysis of the whole coarse dataset with respect to the original. However, there are still details lost during the downsampling process. We believe that we can further preserve some details by improving the Adaptive TFs. One idea that comes to mind is trying to introduce some locality into the resulting Transfer Functions by adding some extra parameter (e.g., gradient length) and extending them to 2D Transfer Functions. Experimenting more on this area will be definitely worth it.

We would also like to extend our architecture to use an out-of-core block-based scheme that allows fetching blocks as needed so that the progressive ray casting can be done for models of larger resolutions not fitting current mobile devices' GPUs.

8.3 Publications

This thesis has generated the following publications:

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez, Frederic Pérez. (May 2015) *Feature-Preserving Downsampling for Medical Images. In EuroVis 2015: The EG/VGTC Conference on Visualization: Posters track. European Association for Computer Graphics (Eurographics)* [15]

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Frederic Pérez, Pere-Pau Vázquez. (June 2016) *Adaptive Transfer Functions. Improved Multiresolution Visualization of Medical Models. The Visual Computer, 32(6-8), 835-845.* [16]

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (July 2017) *Downsampling methods for medical datasets. In Proceedings of the CGVCVIP 2017: International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing: Lisbon, Portugal, (pp. 12-20). IADIS Press.* [17]

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (June 2017) *Downsampling and Storage of Pre-Computed Gradients. In CEIG 2017: XXVII Spanish Computer Graphics Conference: Sevilla, Spain, (pp. 51-60). European Association for Computer Graphics (Eurographics).* [18]

- Jesús Díaz-García, Pere Brunet, Isabel Navazo, Pere-Pau Vázquez. (2018) *Progressive Ray Casting for Volumetric Models on Mobile Devices. Computers & Graphics. DOI: https://doi.org/10.1016/j.cag.2018.02.007* [19]

Bibliography

[1] Aurich, V., and Weule, J. Non-linear gaussian filters performing edge preserving diffusion. In *Mustererkennung 1995, 17. DAGM-Symposium* (London, UK, UK, 1995), Springer-Verlag, pp. 538–545.

[2] Balsa Rodríguez, M., Gobbetti, E., Iglesias Guitián, J. A., Makhinya, M., Marton, F., Pajarola, R., and Suter, S. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum 33*, 6 (2014), 77–100.

[3] Balsa Rodríguez, M., and Vázquez Alcocer, P.-P. Practical volume rendering in mobile devices. In *International Symposium on Visual Computing* (2012), Springer, pp. 708–718.

[4] Bentum, M. J., Lichtenbelt, B. B. A., and Malzbender, T. Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (Sep 1996), 242–254.

[5] Beyer, J., Hadwiger, M., Möller, T., and Fritz, L. Smooth Mixed-Resolution GPU Volume Rendering. In *IEEE/ EG Symposium on Volume and Point-Based Graphics* (2008), H.-C. Hege, D. Laidlaw, R. Pajarola, and O. Staadt, Eds., The Eurographics Association.

[6] Beyer, J., Hadwiger, M., and Pfister, H. A Survey of GPU-Based Large-Scale Volume Visualization. *Proceedings EuroVis 2014* (2014).

[7] Boada, I., Navazo, I., and Scopigno, R. Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer 17*, 3 (2001), 185–197.

[8] Campoalegre, L., Brunet, P., and Navazo, I. Interactive visualization of medical volume models in mobile devices. *Personal and Ubiquitous Computing 17*, 7 (2013), 1503–1514.

[9] Campoalegre, L., Navazo, I., and Crosa, P. B. Gradient octrees: A new scheme for remote interactive exploration of volume models. In *2013 International Conference on Computer-Aided Design and Computer Graphics* (Nov 2013), pp. 306–313.

[10] Congote, J., Segura, A., Kabongo, L., Moreno, A., Posada, J., and Ruiz, O. Interactive visualization of volumetric data with webgl in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology* (2011), ACM, pp. 137–146.

[11] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 15–22.

[12] Csébfalvi, B., and Domonkos, B. Prefiltered gradient reconstruction for volume rendering. *Journal of WSCG 17*, 1 (Jan 2009), 49–56.

[13] Décoret, X., Durand, F., Sillion, F. X., and Dorsey, J. Billboard clouds for extreme model simplification. *ACM Trans. Graph. 22*, 3 (July 2003), 689–696.

[14] Deering, M. Geometry compression. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 13–20.

[15] Díaz-García, J., Brunet, P., Navazo, I., Pérez, F., and Vázquez Alcocer, P.-P. Feature-preserving downsampling for medical images. In *EuroVis, posters* (May 2015).

[16] Díaz-García, J., Brunet, P., Navazo, I., Perez, F., and Vázquez Alcocer, P.-P. Adaptive transfer functions. *Vis. Comput. 32*, 6-8 (June 2016), 835–845.

[17] Díaz-García, J., Brunet, P., Navazo, I., Pérez, F., and Vázquez Alcocer, P.-P. Downsampling methods for medical datasets. In *11th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing (CGVCVIP 2017)* (May 2017).

[18] Díaz-García, J., Brunet, P., Navazo, I., and Vázquez Alcocer, P.-P. Downsampling and storage of pre-computed gradients for volume rendering. In *Congreso Español de Informática Gráfica (CEIG 2017)* (June 2017).

[19] Díaz-García, J., Brunet, P., Navazo, I., and Vázquez Alcocer, P.-P. Progressive ray casting for volumetric models in mobile devices. *Computers and Graphics* (2017).

[20] Duda, R. O., and Hart, P. E. *Pattern Classification and Scene Analysis.* John Wiley and Sons, 1973.

[21] Duffy, B., Carr, H., and Möller, T. Integrating isosurface statistics and histograms. *IEEE Transactions on Visualization and Computer Graphics 19* (February 2013), 263–277.

[22] Elber, G., Chen, X., and Cohen, E. Mold accessibility via gauss map analysis. *Journal of Computing and Information Science in Engineering 5*, 2 (December 2004), 79–85.

[23] Engel, K., Kraus, M., and Ertl, T. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 2001), HWWS '01, ACM, pp. 9–16.

[24] Fisher, M., Dorgham, O., and Laycock, S. D. Fast reconstructed radiographs from octree-compressed volumetric data. *International Journal of Computer Assisted Radiology and Surgery 8*, 2 (2013), 313–322.

[25] Fogal, T., and Krüger, J. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization* (November 2010).

[26] Fogal, T., Schiewe, A., and Krüger, J. An analysis of scalable GPU-based ray-guided volume rendering. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on* (2013), IEEE, pp. 43–51.

[27] Frey, S., Sadlo, F., Ma, K. L., and Ertl, T. Interactive progressive visualization with space-time error control. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (Dec 2014), 2397–2406.

[28] Gobbetti, E., Iglesias Guitián, J., and Marton, F. COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum 31*, 3pt4 (2012), 1315–1324. Proc. EuroVis 2012.

[29] Gobbetti, E., Marton, F., and Iglesias Guitián, J. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (2008), 797–806. Proc. CGI 2008.

[30] GUTENKO, I., PETKOV, K., PAPADOPOULOS, C., ZHAO, X., PARK, J. H., KAUFMAN, A., AND CHA, R. Remote volume rendering pipeline for mHealth applications. vol. 9039, pp. 903904–903904–7.

[31] GUTHE, S., AND STRASSER, W. Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics 28*, 1 (2004), 51–58.

[32] GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. Interactive Rendering of Large Volume Data Sets. In *Proceedings of the Conference on Visualization '02* (Washington, DC, USA, 2002), VIS '02, IEEE Computer Society, pp. 53–60.

[33] HACHAJ, T. Real time exploration and management of large medical volumetric datasets on small mobile devicesevaluation of remote volume rendering approach. *International Journal of Information Management 34*, 3 (2014), 336–343.

[34] HADWIGER, M., KNISS, J. M., REZK-SALAMA, C., WEISKOPF, D., AND ENGEL, K. *Real-time Volume Graphics.* A. K. Peters, Ltd., Natick, MA, USA, 2006.

[35] HARADA, T., AND HOWES, L. Introduction to GPU Radix Sort.

[36] HEGE, H. C., HOLLERER, T., AND STALLING, D. Volume rendering mathematical models and algorithmic aspects, 1993.

[37] JEGX. OpenGL 4.2 Atomic Counters: Rasterization Pattern, Helper for Rendering Optimization (Windows, Linux). `http://www.geeks3d.com`, oct 2013. [Online; accessed 14-December-2016].

[38] KIPFER, P., AND WESTERMANN, R. Improved gpu sorting. In *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, 2005, pp. 733–746.

[39] KITWARE, INC. VES, the VTK OpenGL ES rendering toolkit. http://www.vtk.org/Wiki/VES, November 2014. [Online; accessed 23-March-2017].

[40] KOPF, J., SHAMIR, A., AND PEERS, P. Content-adaptive image downscaling. *ACM Trans. Graph. 32*, 6 (Nov. 2013), 173:1–173:8.

[41] KRATZ, A., REININGHAUS, J., HADWIGER, M., AND HOTZ, I. Adaptive screen-space sampling for volume ray-casting. Tech. Rep. 11-04, ZIB, Takustr.7, 14195 Berlin, 2011.

[42] Kraus, M., and Bürger, K. Interpolating and Downsampling RGBA Volume Data. In *Proceedings of Vision, Modeling, and Visualization 2008* (2008).

[43] Kraus, M., and Ertl, T. Topology-Guided Downsampling. In *Volume Graphics* (2001), K. Mueller and A. Kaufman, Eds., The Eurographics Association.

[44] Krüger, J., and Westermann, R. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), VIS '03, IEEE Computer Society, pp. 38–.

[45] Kwon, K., Kim, M.-S., and Shin, B.-S. A fast 3d adaptive bilateral filter for ultrasound volume visualization. *Computer Methods and Programs in Biomedicine 133* (Sept. 2016), 25 – 34.

[46] LaMar, E., Hamann, B., and Joy, K. I. Multiresolution Techniques for Interactive Texture-based Volume Visualization. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years* (Los Alamitos, CA, USA, 1999), VIS '99, IEEE Computer Society Press, pp. 355–361.

[47] Lamberti, F., and Sanna, A. A solution for displaying medical data models on mobile devices. *SEPADS 5* (2005), 1–7.

[48] Levoy, M. Volume rendering by adaptive refinement. *The Visual Computer 6*, 1 (1990), 2–7.

[49] Levoy, M., and Whitaker, R. Gaze-directed Volume Rendering. *SIGGRAPH Comput. Graph. 24*, 2 (Feb. 1990), 217–223.

[50] Ljung, P., Krüger, J., Gröller, E., Hadwiger, M., Hansen, C. D., and Ynnerman, A. State of the art in transfer functions for direct volume rendering. In *Proceedings of the Eurographics / IEEE VGTC Conference on Visualization: State of the Art Reports* (Goslar Germany, Germany, 2016), EuroVis '16, Eurographics Association, pp. 669–691.

[51] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 163–169.

[52] Max, N. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (June 1995), 99–108.

[53] MOSER, M., AND WEISKOPF, D. Interactive volume rendering on mobile devices. In *In Vision, Modeling, and Visualization VMV 2008 Conference Proceedings* (2008), pp. 217–226.

[54] MOVANIA, M. M., CHIEW, W. M., AND LIN, F. On-site volume rendering with gpu-enabled devices. *Wirel. Pers. Commun. 76*, 4 (June 2014), 795–812.

[55] MOVANIA, M. M., AND LIN, F. High-performance volume rendering on the ubiquitous webgl platform. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on* (2012), IEEE, pp. 381–388.

[56] MOVANIA, M. M., AND LIN, F. Mobile visualization of biomedical volume datasets. *J. Internet Technol. Secur. Trans 1*, 2 (2012), 52–60.

[57] MOVANIA, M. M., AND LIN, F. Ubiquitous medical volume rendering on mobile devices. In *International Conference on Information Society (i-Society 2012)* (June 2012), pp. 93–98.

[58] MOVANIA, M. M., AND LIN, F. Real-time volumetric lighting for webgl. *WebGL Insights* (2015), 261.

[59] NEUMANN, L., CSEBFALVI, B., KÖNIG, A., AND GRÖLLER, M. E. Gradient estimation in volume data using 4d linear regression. Tech. Rep. TR-186-2-00-03, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Feb. 2000. human contact: technical-report@cg.tuwien.ac.at.

[60] NEUMANN, L., CSEBFALVI, B., VIOLA, I., MLEJNEK, M., AND GRÖLLER, M. E. Feature-preserving volume filtering. In *Data Visualization 2002* (May 2002), ACM, pp. 105–114.

[61] NOGUERA, J., AND JIMÉNEZ, J. Visualization of very large 3d volumes on mobile devices and webgl. In *20th WSCG international conference on computer graphics, visualization and computer vision. WSCG* (2012), Citeseer.

[62] NOGUERA, J. M., JIMÉNEZ, J. J., AND OSUNA-PÉREZ, M. C. Development and evaluation of a 3d mobile application for learning manual therapy in the physiotherapy laboratory. *Computers & Education 69* (2013), 96–108.

[63] Noguera, J. M., and Jiménez, J. R. Mobile volume rendering: Past, present and future. *IEEE transactions on visualization and computer graphics 22*, 2 (2016), 1164–1178.

[64] Noguera, J. M., Jiménez, J.-R., Ogáyar, C. J., and Segura, R. J. Volume rendering strategies on mobile devices. In *GRAPP/IVAPP* (2012), pp. 447–452.

[65] Noon, C. J. A Volume Rendering Engine for Desktops, Laptops, Mobile Devices and Immersive Virtual Reality Systems using GPU-Based Volume Raycasting. Master's thesis, Iowa State University, 2012.

[66] Oliveira, J. a. F., and Buxton, B. F. Pnorms: Platonic derived normals for error bound compression. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 2006), VRST '06, ACM, pp. 324–333.

[67] OsiriX Imaging Software. OsiriX HD. http://www.osirix-viewer.com/, November 2014. [Online; accessed 23-March-2017].

[68] Öztireli, A. C., and Gross, M. Perceptually based downscaling of images. *ACM Trans. Graph. 34*, 4 (July 2015), 77:1–77:10.

[69] Paris, S., and Durand, F. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision 81*, 1 (Jan. 2009), 24–52.

[70] Pharr, M., and Randima, F. *GPU Gems 2*. Addison-Wesley Professional, 2005.

[71] Phong, B. T. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June 1975), 311–317.

[72] Raster Images. Oviyam - Web DICOM browser. http://oviyam.raster.in/ioviyam2.html, November 2014. [Online; accessed 23-March-2017].

[73] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel Distributed Processing* (May 2009), pp. 1–10.

[74] Schiewe, A., Anstoots, M., and Krger, J. State of the Art in Mobile Volume Rendering on iOS Devices. In *Eurographics Conference on Visualization (EuroVis) - Short Papers* (2015), E. Bertini, J. Kennedy, and E. Puppo, Eds., The Eurographics Association.

[75] SCHULTZ, C., AND BAILEY, M. Interacting with large 3d datasets on a mobile device. *IEEE Computer Graphics and Applications 36*, 5 (2016), 19–23.

[76] SHIRLEY, P., AND TUCHMAN, A. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph. 24*, 5 (Nov. 1990), 63–70.

[77] SICAT, R., HADWIGER, M., KRÜGER, J., AND MÖLLER, T. Sparse PDF volumes for consistent multi-resolution volume rendering. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Visualization) 20*, 12 (2014), 2417–2426.

[78] SMITH, S. M., AND BRADY, J. M. Susan&mdash;a new approach to low level image processing. *Int. J. Comput. Vision 23*, 1 (May 1997), 45–78.

[79] SOUSA, R., NISI, V., AND OAKLEY, I. Glaze: A visualization framework for mobile devices. In *Human-Computer Interaction–INTERACT*. Springer, 2009, pp. 870–873.

[80] THELEN, S., MEYER, J., EBERT, A., AND HAGEN, H. Giga-scale multiresolution volume rendering on distributed display clusters. In *Human Aspects of Visualization*. Springer, 2011, pp. 142–162.

[81] TOMASI, C., AND MANDUCHI, R. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)* (Jan 1998), pp. 839–846.

[82] WANG, C., GAO, J., LI, L., AND SHEN, H.-W. A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Proceedings of the Fourth Eurographics/IEEE VGTC conference on Volume Graphics* (2005), Eurographics Association, pp. 11–19.

[83] WANG, L., MENG, Z., YAO, X. S., LIU, T., SU, Y., AND QIN, M. Adaptive speckle reduction in oct volume data based on block-matching and 3-d filtering. *IEEE Photonics Technology Letters 24*, 20 (Oct 2012), 1802–1804.

[84] WANG, Y. S., WANG, C., LEE, T. Y., AND MA, K. L. Feature-Preserving Volume Data Reduction and Focus+Context Visualization. *IEEE Transactions on Visualization and Computer Graphics 17*, 2 (Feb 2011), 171–181.

[85] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on 13*, 4 (April 2004), 600–612.

[86] Weber, N., Waechter, M., Amend, S. C., Guthe, S., and Goesele, M. Rapid, detail-preserving image downscaling. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 205:1–205:6.

[87] Weiler, M., Westermann, R., Hansen, C., Zimmermann, K., and Ertl, T. Level-of-detail volume rendering via 3D textures. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization* (New York, NY, USA, 2000), VVS '00, ACM, pp. 7–13.

[88] Xin, Y., and Wong, H. C. Intuitive volume rendering on mobile devices. In *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)* (Oct 2016), pp. 696–701.

[89] Xu, X., Sakhaee, E., and Entezari, A. Volumetric data reduction in a compressed sensing framework. *Computer Graphics Forum 33*, 3 (2014), 111–120.

[90] Younesy, H., Möller, T., and Carr, H. Improving the quality of multi-resolution volume rendering. In *Proc. Joint Eurographics/IEEE VGTC conference on Visualization* (2006), Eurographics Association, pp. 251–258.