IMPROVED ALGORITHMS FOR VOLUME RENDERING AND MESH PROCESSING

André de Almeida Maximo

Doctoral Thesis presented to the Graduation Program in Systems Engineering and Computer Science, COPPE, from the Universidade Federal do Rio de Janeiro, as a partial fulfillment of the requirements for the degree of Doctor in Systems Engineering and Computer Science.

Advisors: Ricardo Cordeiro de Farias
Amitabh Varshney

Rio de Janeiro
July, 2010

IMPROVED ALGORITHMS FOR VOLUME RENDERING AND MESH PROCESSING

André de Almeida Maximo

THESIS SUBMITTED TO THE INSTITUTE ALBERTO LUIZ COIMBRA OF GRADUATION IN ENGINEERING (COPPE) FROM THE UNIVERSIDADE FEDERAL DO RIO DE JANEIRO AS A PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR IN SYSTEMS ENGINEERING AND COMPUTER SCIENCE.

Examined by:

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. Claudio Esperança, Ph.D.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Amitabh Varshney, Ph.D.

Dr. Diego Fernandes Nehab, Ph.D.

Prof. João Luiz Dihl Comba, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
JULY, 2010

Maximo, André de Almeida

Improved Algorithms for Volume Rendering and Mesh Processing/André de Almeida Maximo. – Rio de Janeiro: UFRJ/COPPE, 2010.

XII, 103 p.: il.; 29, 7cm.

Advisors: Ricardo Cordeiro de Farias
Amitabh Varshney

Thesis (doctorate) – UFRJ/COPPE/Program in Systems Engineering and Computer Science, 2010.

Bibliography: p. 94 – 103.

1. Computer Graphics. 2. Volume Rendering. 3. Mesh Processing. I. Farias, Ricardo Cordeiro de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Program in Systems Engineering and Computer Science. III. Title.

To my dear parents and family,
for their support and love

Acknowledgments

I would like to thank everyone who contributed to the completion of this thesis.

Professors of the Laboratory for Computer Graphics (LCG): Ricardo Farias, Ricardo Marroquim, Claudio Esperança, Paulo Roma and Antonio Oliveira. For the questions answered and support always present. In especially the Ricardos (Farias and Marroquim) for their friendship and chats in important decisions.

Professor Amitabh Varshney of the Graphics and Visual Informatics Laboratory (GVIL) for having hosted me as an intern at the University of Maryland (UMD) in 2009, on my 1 year of J-1 Internship program. And by his patience and wisdom on our many meetings where we talk about research and philosophy.

All my friends from LCG who accompanied me through my doctorate: Álvaro "Bubu", "Dino" Saulo, Ricardo "Rico", Yalmar, Wagner, Fláv-IO, Felipe "Cabeludo", Leandro "Brucutu" and many others; for their conversations, discussions and advices needed academically and personally.

All my friends from GVIL who accompanied the struggle during my internship: Robert Patro "Rob", Sujal Bista and Yiu Ip "Horace". For their help and conversations on the countless hours of lab work.

I thank my friends in childhood, of many years and known recently: André, Anderson, Nelson, Jô, Eneida, Emilian, Danilo, Maise, Thatiana, Melissa, Bruna, Vanessa, Léo Claudino, Aninha, Joao, William, Léo Mineiro, Mandy and many others; for their strength, friendship and encouragement over the past years.

I thank those responsible for leisure time: Thirsty Turtle, U Street, Lapa, Irish Pub, Cinemark, Wizards of the Coast, Joanne K. Rowling, Dan Brown and Bernard Cornwell. These moments so important for the continuity of my work.

I thank my family: Grandma Nazita, Michel, Fabiano, Andréia, Cristiano, Aunt Terezinha, Aunt Celinha, Uncle Tercílio, Uncle Tuninho, and many others; for all the help and important presence in my life.

I thank also my siblings: Mário and Barbara; and my parents: Paulo and Magda; for their support, presence and confidence essential for the completion of this thesis.

I thank the Brazilian agency CNPq (National Counsel of Technological and Scientific Development) for the financial support during my doctorate and internship; and Vicente Batista for the LaTeX class CoppeTeX used to write this thesis.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

IMPROVED ALGORITHMS FOR VOLUME RENDERING AND MESH PROCESSING

André de Almeida Maximo

July/2010

Advisors: Ricardo Cordeiro de Farias
Amitabh Varshney

Program: Systems Engineering and Computer Science

In this thesis, improved algorithms are presented for volume rendering and mesh processing. In the research area of volume rendering, the goal is to improve computational performance and memory consumption, using programmable graphics cards, while in the area of mesh processing, the goal is to introduce a method to augment the usage of shape similarity of a surface. The volume rendering algorithms are based in ray casting and cell projection, handling both regular and irregular data, and employing both direct and indirect volume rendering techniques. The mesh processing method, on the other hand, augments the usage of self-similarity of models to propagate processing done in one part of the mesh to many others similar parts. The presented mesh processing method is used in two distinct applications: detail transfer and parameterization. Finally, the volume rendering techniques are compared against each other and appropriate state-of-the-art techniques.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

ALGORITMOS APRIMORADOS PARA VISUALIZAÇÃO VOLUMÉTRICA E PROCESSAMENTO DE MALHAS

André de Almeida Maximo

Julho/2010

Orientadores: Ricardo Cordeiro de Farias
Amitabh Varshney

Programa: Engenharia de Sistemas e Computação

Nesta tese são apresentados algoritmos aprimorados para visualização volumétrica e processamento de malhas. Na área de pesquisa de visualização volumétrica, o objetivo é a melhoria de desempenho computacional e consumo de memória, usando placas gráficas programáveis, enquanto na área de processamento de malhas o objetivo é introduzir um método para ampliar o uso de similaridade de formas de uma superfície. Os algoritmos de visualização se baseiam em traçado de raios e projeção de células, tratando tanto dados volumétricos regulares quanto irregulares e renderizando os dados tanto diretamente quanto indiretamente através de isosuperfícies. O método de processamento de malhas, por outro lado, amplia o uso de auto-similaridade de modelos para propagar processamento feito em uma parte do modelo para outras partes similares. O método de processamento de malhas apresentado é usado em duas aplicações distintas: transferência de detalhe e parametrização. E, finalmente, as técnicas de visualização volumétrica são comparadas entre si e a técnicas apropriadas do estado da arte.

Contents

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Volume Rendering 2
1.2 Mesh Processing 5
1.3 GPU Programming 7

2 Related Work **11**

2.1 Volume Rendering 12
2.1.1 Volume Rendering Integral 12
2.1.2 Visibility Ordering 15
2.1.3 Ray Casting 15
2.1.4 Cell Projection 17
2.2 Mesh Processing 19
2.2.1 Reflective Symmetries 19
2.2.2 Similarity Descriptors 20

3 Volume Rendering **23**

3.1 VF-Ray-GPU 24
3.2 RPTINT 31
3.3 IPTINT 36
3.4 HAPT 41

4 Mesh Processing **48**

4.1 SAMPLE 49
4.2 Applications 57

5 Results **60**

5.1 Volume Rendering 61
5.2 Mesh Processing 75

6 Conclusions **79**

A Developed Algorithms **81**

B VF-Ray Algorithm **82**

C PT Algorithm **85**

D PTINT Algorithm **88**

Bibliography **94**

List of Figures

1.1 Example of medical images . . . 2
1.2 Example of direct volume rendering . . . 3
1.3 Example of editing the transfer function . . . 4
1.4 Example of indirect volume rendering . . . 5
1.5 Example of mesh processing . . . 6
1.6 Graphics pipeline . . . 8
1.7 Processing scheme using CUDA . . . 9

2.1 Model of the volume rendering integral . . . 12
2.2 Simplified model of the volume rendering integral . . . 13
2.3 Example of ray casting . . . 16
2.4 Types of faces . . . 16
2.5 Example of cell projection . . . 17
2.6 Example of reflection plane . . . 19
2.7 Example of the similarity signature of a vertex . . . 21

3.1 Ray coherence of VF-Ray . . . 24
3.2 Kernels of VF-Ray-GPU . . . 26
3.3 Data structures of VF-Ray-GPU . . . 27
3.4 Hash table of VF-Ray-GPU . . . 29
3.5 Thread scheme of the third kernel of VF-Ray-GPU . . . 30
3.6 Overview of RPTINT . . . 31
3.7 Pipeline of RPTINT . . . 33
3.8 Arrays structure of RPTINT . . . 34
3.9 Textures scheme of IPTINT . . . 36
3.10 I/O of the first step of IPTINT . . . 37
3.11 Array structures of IPTINT . . . 38
3.12 Iso-surface rendering of IPTINT . . . 40
3.13 HAPT's Framework . . . 42
3.14 HAPT's pipeline . . . 42
3.15 Example of class 1 projection of PT . . . 45

4.1 Example of similarity space of SAMPLE 48
4.2 Similarity descriptor example of SAMPLE 50
4.3 Zernike expansions for heightmaps in SAMPLE 52
4.4 Differences (I) between similarity methods 54
4.5 Differences (II) between similarity methods 55
4.6 Illustration of the primal and dual spaces of SAMPLE 56
4.7 SAMPLE application: mesh parameterization 58
4.8 SAMPLE application: detail transfer 59

5.1 Images generated by VF-Ray-GPU 64
5.2 Images generated by RPTINT 65
5.3 Image from the “spx+” dataset generated by IPTINT 66
5.4 Images generated (I) by IPTINT 67
5.5 Images generated (II) by IPTINT 68
5.6 Volumetric data “torso” rendered with iso-surfaces by HAPT 70
5.7 Volumetric data “torso” rendered without iso-surfaces by HAPT . . . 71
5.8 Time-varying volume rendering 72
5.9 Images generated by HAPT 73
5.10 Dual space example of SAMPLE 75
5.11 Symmetry example of SAMPLE 76
5.12 Application example of the immediate dual neighbor 76
5.13 Nearest similarities example 77

List of Tables

3.1 Error between centroid sorting and MPVONC. 44

5.1 Properties of the tested datasets of VF-Ray-GPU 62
5.2 Memory aspects of the algorithms 63
5.3 Memory and timing comparison 63
5.4 Comparison of the original VF-Ray and VF-Ray-GPU 63
5.5 Performance measurement of RPTINT 64
5.6 Timing of the third and fourth step of RPTINT 66
5.7 Comparison of different algorithms with IPTINT 67
5.8 Performance measurement of IPTINT 68
5.9 Performance measurement of HAPT 69
5.10 Comparison of different algorithms (I) with HAPT 70
5.11 Comparison of different algorithms (II) with HAPT 72
5.12 Time spent for establishing the dual space in SAMPLE 78

Chapter 1

Introduction

"Think of giving not as a duty
but as a privilege."
– John Davison Rockefeller

This thesis presents several improved algorithms for volume rendering and mesh processing. The rendering techniques presented here aim to improve the computational performance and memory consumption, while the mesh processing method introduces a new concept in the usage of self-similarity of models. In rendering, the algorithms presented target the visualization of volumetric datasets, specifically scalar fields defined in regular or irregular grids, using programmable graphics cards (GPUs). In mesh processing, the method introduced uses non-local neighborhood defined by mesh descriptors to propagate mesh processing done in one part of the model to many other parts, which share some desired feature. The algorithms presented of these two research areas are distinct in essence (they are listed in Appendix A).

In this chapter a brief introduction on the topics related to this thesis is presented. In the next chapter (2), previous works covering both areas are discussed. The contributions of this thesis in the area of volume rendering are presented in Chapter 3, and in the area of mesh processing in Chapter 4. The results obtained in both areas are presented in Chapter 5, and the conclusions of the work in Chapter 6. Additionally, the Appendices provide complementary information for the understanding of the material presented here.

1.1 Volume Rendering

The volume rendering area is responsible for generating 2D visualizations from 3D data, also called volumetric data. The main source for this type of data are numerical simulations of natural phenomena, e.g. Computational Fluid Dynamics (CFD), and measurement devices, e.g. Magnetic Resonance Imaging (MRI) and Computed Tomography (CT) in medicine and Seismic Tomography in geology. Generally, measurement devices produce regular volume data, while simulations yield both regular and irregular data.

An example of regular volume data in medical images can be seen in Figure 1.1. These images are from a CT scan of a human skull and constitute the raw data analyzed by physicians. The CT scan images comprise a sequence of 2D images, or slices, which together form a 3D image or volume.

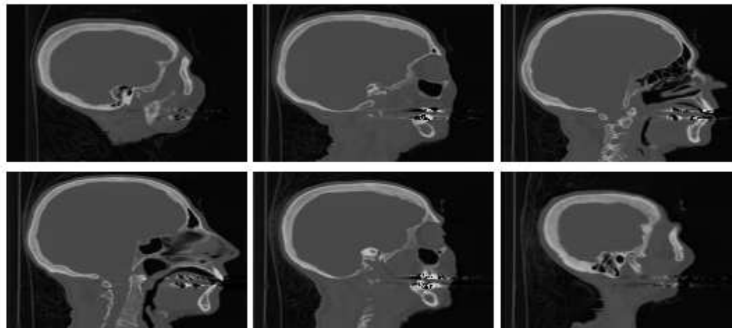


Figure 1.1: Images from a Computed Tomography of a human skull with $256 \times 256 \times 113$ resolution, extracted from the YZ plane. This volume is the *Cadaver Head data* from the UNC Chapel Hill Volume Rendering Test Data Set.

The regular volume, in this case a skull, is the object of interest, which is analyzed slice by slice using the images shown in Figure 1.1. These images were generated from a real dataset, using an application developed by me available at:

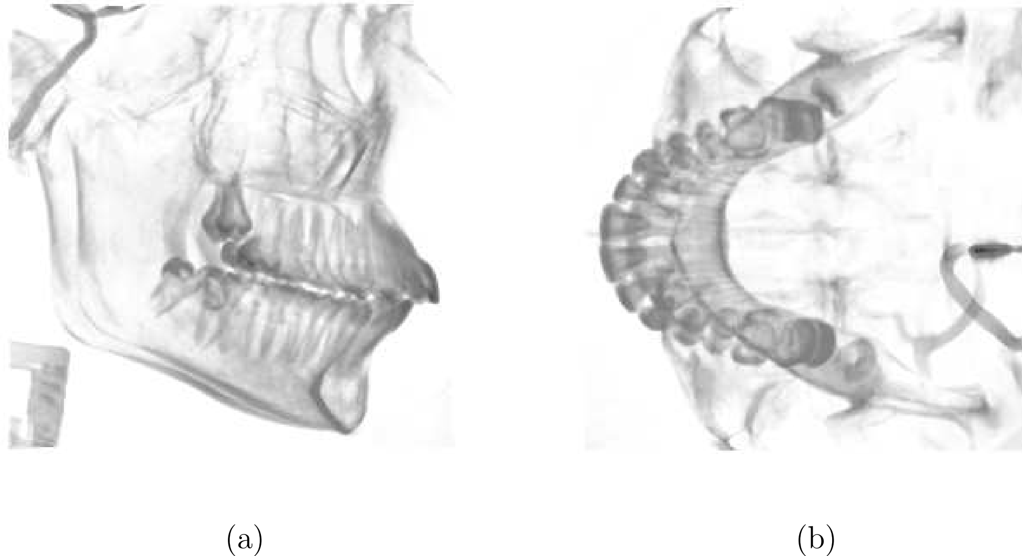
http://code.google.com/p/image3dviewer [1].

The main disadvantage of this type of 2D visual analysis is the lack of a three-dimensional component, important for approximating the volumetric data to the real object. The volume rendering area aims to add this component, generating images of the volume from any viewpoint and composing the slices to allow the visual sensation of depth.

One example of volume rendering, using one of the techniques presented in this thesis (explained in Section 3.2), can be seen in Figure 1.2. The volume used to generate these images is defined by the same slices shown in Figure 1.1, but composing

[1]Source code of applications related to this thesis are open source under the *GNU General Public License version 3* hosted at Google™ code.

the slices using transparency. The regular data in this example is a scalar field that represents density values of the human skull, where the gray scale corresponds to the samples of the scalar field.



(a) (b)

Figure 1.2: Images generated using volume rendering of the human skull shown in Figure 1.1 for two different viewpoints: side (a); and top (b).

One of the challenges in this type of rendering is the interactivity, that is, allowing the efficient visualization of the volume by changing the viewpoint. Depending on the size of the data and the visualization window, the time spent in rendering can be very long (seconds or even minutes) compromising the interaction with the volume. For example, the application used in Figure 1.2 allows the interaction in real time (using a common off-the-shelf computer) of 70 frames per second (fps) for regular data with $64 \times 64 \times 64$ voxels and a windows size of 512×512 pixels. However for datasets of 256^3 voxels or more, the performance drops to less than 1 fps, compromising the visual feedback of the interaction. This performance challenge is important in the area of volume rendering, where the volume data are generally large and require an extensive analysis.

Another way of visualizing the volumetric data is based on the control of the transparency and the highlighting of the volume. This control is done by means of the transfer function, responsible for mapping scalar values to color and opacity. Through this function, for example, parts of the volume can be hidden allowing the visualization of regions of interest with more details. Figure 1.3 exemplifies the usage of the transfer function to reduce the opacity of small scalar values, coloring the volume from blue to red. The volumetric data used in this example is irregular and comprises a fluid simulation of air flow over an aircraft wing. The left window is the volume rendering itself, using one of the algorithms presented in this research

work (explained in Section 3.4), while the right window exhibits the interface to edit the transfer function. This windows pattern is used by some of the applications available with this thesis.

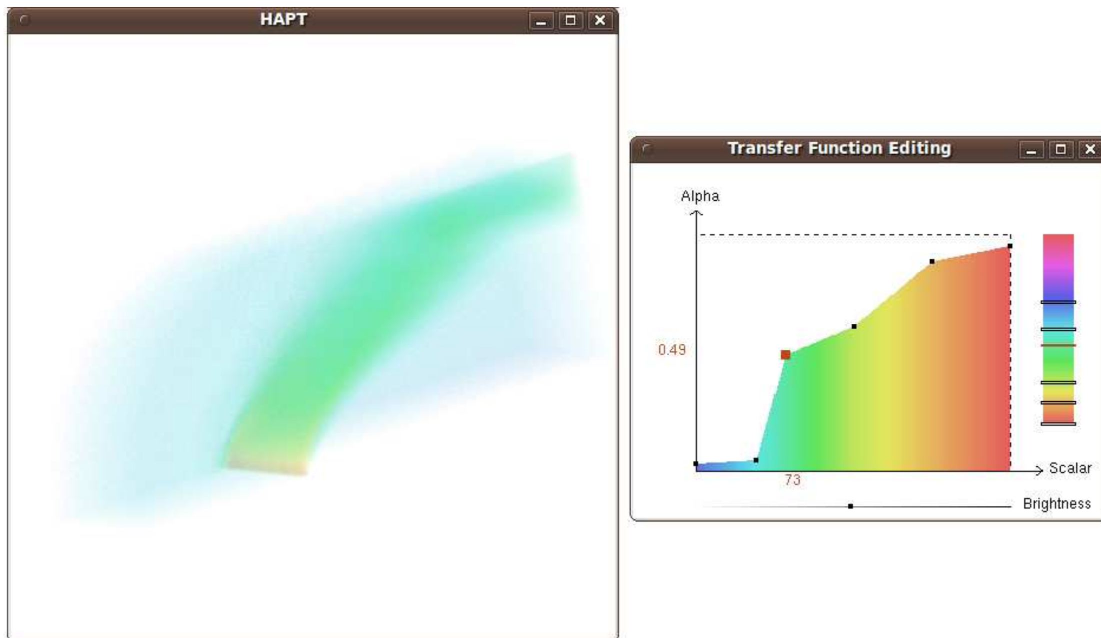


Figure 1.3: Example of editing the transfer function. The visualization of the volumetric data (left) is modified by the editing of the transfer function associated with it (right). This volume is the *Blunt Fin* dataset from the NASA Advanced Supercomputing Division.

Just as the performance in manipulating the viewpoint is important in volume rendering, the efficient editing of the transfer function is also interesting. This type of interaction is accomplished if, while the function is modified, the volume reflect, the changes simultaneously. The modification of the transfer function is given by the change of color values and opacity (the alpha axis in Figure 1.3) associated with the scalar values of the volume. The high performance in this interaction allows a better handling of the volumetric data, enabling data with a wide range of values to be analyzed more effectively.

In contrast with the volume rendering shown so far, known as *direct volume rendering*, there is an alternative rendering of volume data called *indirect volume rendering*. The direct approach is based on rendering the volume as a semitransparent material, while the indirect approach seeks to find and render surfaces of equal value within the volume, called *iso-surfaces*. Figure 1.4 shows an example of rendering iso-surfaces of a volumetric data representing the spatial probability of electron distribution. The technique used in this rendering is part of this thesis (explained in Section 3.3).

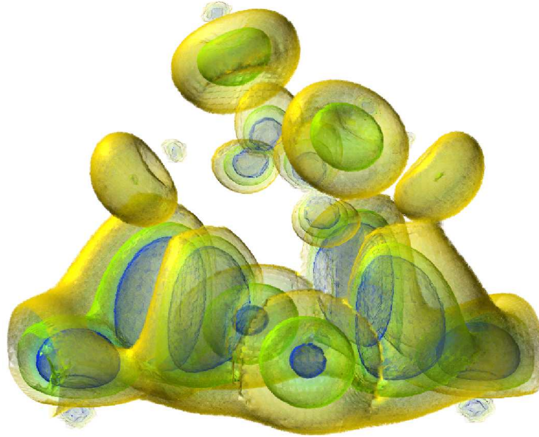


Figure 1.4: Example of indirect volume rendering. The surfaces with the same scalar value, called iso-surfaces, are highlighted inside the volume showing distribution of electron densities. This volume is the *Negative High Potential* (neghip) dataset from the VolVis distribution of SUNY Stony Brook, NY, USA.

The research contributions of this thesis in the area of volume rendering involve both direct and indirect approaches, as well as dealing with both regular and irregular data.

1.2 Mesh Processing

The mesh processing area, a part of the more general area of *geometry processing*, covers algorithms related to acquisition, reconstruction, analysis, storage, retrieval, manipulation, simulation, and transmission of three-dimensional objects, or models. The 3D models covered in this extensive area are, generally, triangular meshes describing the surface of an object of interest. This description is given by a set of vertices and faces forming the discrete surface of the object. Examples of 3D models can be found in games and movies that use computer graphics, where the virtual objects in scenes are modeled by an artist or scanned from real-world objects using a 3D scanner device.

Figure 1.5 shows an example of a 3D model and of a mathematical technique used in mesh processing applied to part of the example model. The model shown is the *XYZ RGB Asian Dragon*, a sculpture of an Asian dragon made available by XYZ RGB Inc. This model contains 108 K vertices and 216 K triangular faces. The technique shown “flattens” a piece of the surface around the selected vertex (shown in red). This technique is called parameterization of the surface and aims to get a 2D representation of the surface embedded in 3D. One of the many application of this technique is the texturing of objects, in which a 2D texture image can be mapped to the surface of a 3D object.

Figure 1.5: Example of a mesh processing technique shown on the Asian Dragon from XYZ RGB Inc. The surface around the selected vertex (in red) is parameterized in two dimensions and shown in the bottom-right corner.

The rendering of the object shown in Figure 1.5 and the window showing the parameterization of the surface are part of this thesis. The data chosen for testing in this research work are scanned data, described by a list of vertices and triangular faces composing the object. From this basic information, other information can be computed, such as the surface normal and curvature. Information related to the 3D model, both basic and inferred computationally, are used in this thesis to analyze the similarity of the meshes, explained in details in Chapter 4.

A major challenge in the area of mesh processing is how to handle the information about the object efficiently and objectively. Complex and compact data structures can save memory but have a negative impact in algorithms' performance. In a similar way, large and complete data structures can facilitate the access to information but consume a prohibitive amount of memory. This concern in structuring the data of a model is governed by how to succinctly and effectively describe local neighborhoods of mesh elements, such as vertices and edges. For example, a given data structure may store for each vertex which vertices are its neighbors, and for each face which faces are its neighbors. This type of local data structure is important for the majority of mesh processing techniques.

Another way to think about the data structures for 3D models is globally. A global, or non-local, data structure can be used to complement a local data structure, allowing new concepts in mesh processing algorithms. For example, a data structure can store for each vertex the vertices which are most similar to it, given some similarity criteria, and mesh processing techniques can take advantage of this additional information. The vast majority of objects, scanned from the real world

or artificially modeled, have regions with nearly identical properties, such as color, shape or texture. These properties can be used as a similarity criteria when building a non-local data structure.

The research contributions of this work in the area of mesh processing refer to the usage of repeated patterns in order to propagate processing done in one region to other similar regions in the mesh.

1.3 GPU Programming

The research work of this thesis related to volume rendering is based on GPU programming. A programmable graphics processor, or simply GPU, enables the developer to access resources and run programs on a massively-parallel processing unit, providing significant performance compared to conventional CPU programming. The main difference in programming lies in the concept of the processor: while the CPU executes instructions sequentially with high processing control and has several levels of cache to reduce memory latency; the GPU executes instructions in a parallel streaming model on a large amount of data while hiding memory latency by using several processing cores.

Both in scientific and commercial venues, GPU programming is currently being used by developers beyond computer graphics. Problems that require high-performance computing can use the GPU as a massively-parallel coprocessor of the CPU. In the case of the GPU being used for graphics, the developer works with shaders and respects the graphics pipeline. In the case of the GPU being used as a generic coprocessor of the CPU, the developer works with kernels and uses the multiprocessors. The concept of programming kernels is more easy to understand than the concept of programming shaders; the first allows general processing while the second is restricted to the part of the graphics pipeline it runs. Regardless of the GPU running a shader or a kernel, all multiprocessors can be used in the task. This management of computing resource of the graphics card was introduced by the so-call unified shader model or architecture.

The shaders can operate in different parts of the graphics pipeline. Figure 1.6 shows a summary of the stages of the pipeline in accordance with the unified shader model (version 4.0). First, a CPU application sends vertices of geometric primitives, e.g. points or triangles, to the graphics card (I). Then, several operations per vertex are performed in parallel by the graphics card multiprocessors. In this step, a **vertex shader** can be used to replace the fixed functionality in hardware which performs geometric transformations, illumination computation, etc. After this shader, the transformed vertices (II) are sent to the **geometry shader** which performs the primitive assembly. In this step, primitives can be generated, e.g. using the primi-

tives sent by the previous step, or excluded, e.g. by the face culling operations. The generated primitives are then rasterized (a), a process which fills the primitives with pixels, more correctly called pre-pixels or fragments, since these still do not form the final pixels of the framebuffer.

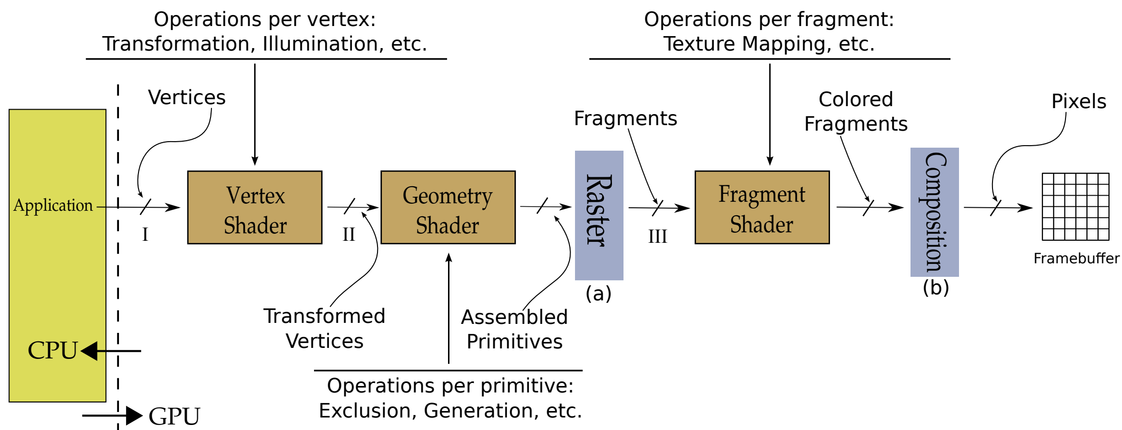


Figure 1.6: Summary of the graphics pipeline of the GPU.

For each new stage of the pipeline, the output is fed back to the input and all the multiprocessors of the GPU are employed. Before the unified shader architecture, each stage of the pipeline constitutes a part of the GPU's multiprocessors and the pipeline was delineated in hardware. Nowadays the concept of the graphics pipeline is abstract, not being explicitly present in the GPU architecture anymore.

The fragments generated by the rasterization (Figure 1.6a) are sent to all the multiprocessors again, where a **fragment shader** can be used (III). In this step, the fixed functionality performs, for example, the texture mapping, where texels (texture elements) are read from the graphics card memory and mapped to the respective fragments according to their texture coordinates. Finally, the colored fragments, by texture or simple color attributes, are sent to the composition process (b) responsible for aggregating them into a matrix of pixels, called the frame buffer. The fragments falling on the same pixel can be composed by a blending function or discarded by a depth function. The content of the frame buffer is usually shown in the graphics window of the application.

The graphics card memory, also called texture memory, can be accessed by any of the three shaders. The memory access by the shaders is restricted to be read-only and suffers a high latency between request and data acquisition. For this reason, the *arithmetic intensity*, a concept defined by the ratio of arithmetic operations per memory access, must be maximized to achieve high-performance computation when using GPU programming.

Examples of source code in GLSL – *OpenGL* [1] *Shading Language* [2] – and how each shader works and the relationship between them are presented in the tutorial: *Introduction to GPU Programming with GLSL* [3]. Besides this article, several shader source codes in GLSL are available at:

http://code.google.com/p/glsl-intro-shaders.

In contrast with GPU programming using shaders, the CUDA – *Compute Unified Device Architecture* [4] – technology for example, allows the development of generic applications using the graphics card as a coprocessor of the CPU. In CUDA, the implementation is done through kernels which are executed by multiple threads. The threads are grouped in blocks, where they share the resources of one GPU's multiprocessor, as can be seen in Figure 1.7. Each multiprocessor has a SIMD architecture (Single Instruction Multiple Data) that performs the same instructions of the kernel, but on different data. Similar to kernels, shaders perform the same instructions for multiple vertices, geometries, or fragments.

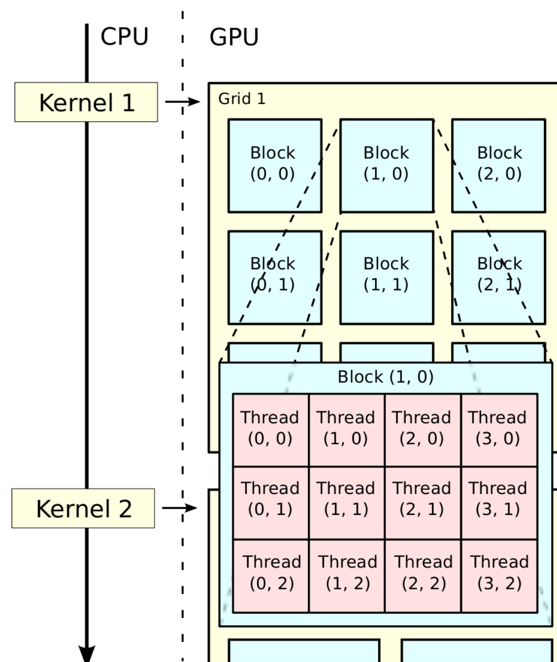


Figure 1.7: Processing scheme using CUDA. Each kernel is associated with a processing grid consisting of blocks, which in turn consists of threads.

A block of threads is an abstraction for a parallel processing unit of the GPU, called multiprocessor, and is grouped by grids. A grid, in turn, abstracts the GPU itself, possessing different multiprocessors. Each kernel program is executed by setting the number of blocks per grid and the number of threads per block. For a new kernel to be executed after the first one, a new grid is instantiated (see example kernels 1 and 2 in Figure 1.7). This programming model simplifies the development of massively-parallel applications, enabling a level of execution control between programming by using shaders and implementing on the CPU.

In this thesis, the improved algorithms for volume rendering are based on the GPU and use GLSL as the programming language for shaders and *C for CUDA* for general-purpose kernels.

Chapter 2

Related Work

"Even if you are on the right track,
you'll get run over if you just sit there."
– Will Rogers

In this chapter we shall review two major areas of computer graphics: volume rendering and mesh processing.

In the volume rendering area, the volume integration model and visibility ordering are reviewed and important algorithms of ray casting and cell projection are considered. The techniques shown here are used as base and/or for comparison by the algorithms presented in this thesis.

In the mesh processing area, algorithms devoted to analysis of surfaces and geometry processing applications are discussed. Descriptors and data structures presented here reinforce the contribution of the methods presented in this thesis.

The research studies reviewed are divided as follows:

- Section 2.1 covers the volume rendering area, with Subsections 2.1.1, 2.1.2, 2.1.3 and 2.1.4 discussing different methods related to the research of this thesis;

- Section 2.2 focuses on works in the mesh processing area, with Subsections 2.2.1 and 2.2.2 discussing different techniques related to the contributions presented here.

2.1 Volume Rendering

In this section we review techniques from the volume rendering area related to the algorithms presented in the next chapter. This section is divided in four parts: in Subsection 2.1.1, integration models for the light interaction with the volume are presented; in Subsection 2.1.2, volume rendering techniques related to visibility ordering are discussed; in Subsection 2.1.3, the ray-casting method is outlined and algorithms based on this method are explained; finally in Subsection 2.1.4, the cell-projection method together with algorithms for this method related to the research work of this thesis are elucidated.

2.1.1 Volume Rendering Integral

Evaluating the physical interaction of light with the volumetric data requires the computation of the volume rendering integral (see Equation 2.1). This integral is an equation to compute the color resulting from the light passing through the volume. MAX [5] presents different models for the interaction of light with the volume in the direct volume rendering area. In this research work, we use the absorption plus emission model for both cell-projection and ray-casting algorithms. Max shows the step-by-step composition of the equation until achieving the volume rendering integral:

$$I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt} ds. \tag{2.1}$$

Through this equation the change of the light ray intensity I from the end of the volume $s = 0$ to the observer $s = D$ is evaluated (see Figure 2.1). The light ray traverses a distance D to the observer, where the first term represents the amount of incoming light I_0, attenuated exponentially by the distance D. The second term adds the amount of light emitted by each point along the path of the ray, taking into account the amount attenuated from the point to the end of the ray.

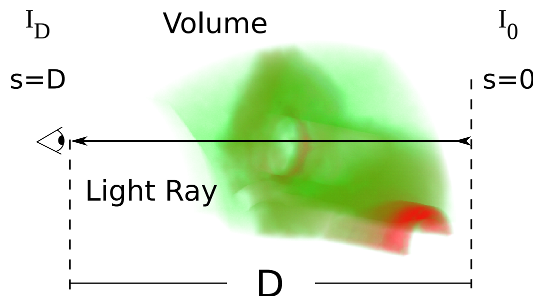


Figure 2.1: Model of the volume rendering integral, with the light ray covering a distance D from the end of the volume to the observer.

The evaluation of this integral frame-by-frame for all the image pixels is a computationally expensive process. Some papers [6–9] improve the performance of their rendering method by simplifying this integral. The model proposed by these methods differs from the Equation 2.1 (see Figure 2.2). In this model, the viewing ray is used (from the observer to the end of the volume) rather than the light ray. Note that the integral depends only on the distance l (*length*) traversed inside each *volume cell*, called *thickness*, and the scalar values of the ray's entry point s_f (*scalar front*) and exit point s_b (*scalar back*). The result of the integral then provides the cell's contribution to the illumination of the pixel.

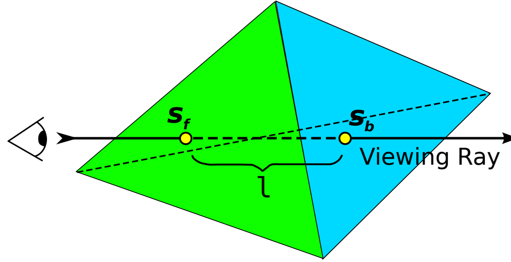


Figure 2.2: Simplified model of the volume rendering integral, using only the scalar front s_f, back s_b and the thickness l of each volume cell.

In a simplified model, the pixel's color can be computed by the average of the color front $C(s_f)$ and back $C(s_b)$, as expressed by SHIRLEY and TUCHMAN [6]:

$$C = \frac{C(s_f) + C(s_b)}{2}. \tag{2.2}$$

Likewise, the opacity α can be calculated using the average of the extinction coefficients of front $\tau(s_f)$ and back $\tau(s_b)$:

$$\alpha = 1 - e^{-\frac{\tau(s_f)+\tau(s_b)}{2}\, l}. \tag{2.3}$$

The colors and extinction coefficients are directly associated with the scalar values of front s_f and back s_b via the transfer function [10], which defines $\tau()$ and $C()$ in Equations 2.2 and 2.3.

The final color and opacity (RGBA) of the pixel, represented by I in Equation 2.1, are computed by the combination of the cells traversed by the same ray. For each cell after the first one, the updated color and opacity C_{i+1} and α_{i+1} are the linear combination of the previous colors C_i and C_{i-1}, and opacities α_i and α_{i-1}, using the following equations:

$$C_{i+1} = C_{i-1} + (1 - \alpha_{i-1})\, C_i, \tag{2.4}$$

$$\alpha_{i+1} = \alpha_{i-1} + (1 - \alpha_{i-1})\, \alpha_i. \tag{2.5}$$

Another way to compute color and opacity is using a linear interpolation between the front and back values. Considering an orthogonal space and the integration performed along the z axis, BUNYK *et al.* [11] express color and opacity with the two following functions:

$$C(z) = \frac{(z_b - z)C(s_f) + (z - z_f)C(s_b)}{\Delta_z}, \tag{2.6}$$

$$\alpha(z) = \frac{(z_b - z)\alpha(s_f) + (z - z_f)\alpha(s_b)}{\Delta_z}. \tag{2.7}$$

In these linear functions, the value of $C(z)$ and $\alpha(z)$ correspond to the value of color and opacity at point z along the path of the ray. These functions should be integrated from z_f (z *front*) to z_b (z *back*) in order to obtain C_{i+1} and α_{i+1}, i.e. color and opacity accumulated in the updated step:

$$C_{i+1}(z) = C_i + \int_{z_f}^{z} C(z)(1 - \alpha(z))dz, \tag{2.8}$$

$$\alpha_{i+1}(z) = \alpha_i + \int_{z_f}^{z} \alpha(z)dz. \tag{2.9}$$

Note that in this simplified model, used by BUNYK *et al.* [11], the opacity is calculated by linear interpolation, that is, the exponential attenuation of Equation 2.1 is not considered. Solving the integrals of Equations 2.8 and 2.9 analytically, they get to the two following equations for the computation of color and opacity during the ray integration:

$$C_{i+1} = C_i + \frac{1}{2}(C_f + C_b)(\alpha_f - 1)\Delta_z - \frac{1}{24}(3C_f\alpha_f + 5C_b\alpha_f + C_f\alpha_b + 3C_b\alpha_b)\Delta_z^2, \tag{2.10}$$

$$\alpha_{i+1} = \alpha_i + \frac{1}{2}(\alpha_f + \alpha_b)\Delta_z. \tag{2.11}$$

A workaround to evaluate the volume rendering integral frame-by-frame is to compute it in advance, storing the discrete results in a table. This technique, called *pre-integration*, was introduced in the context of GPU programming by RÖTTGER *et al.* [8].

One disadvantage of using the pre-integration technique is that if the transfer function changes, the application needs to recalculate the entire table of integration values and store it again. This procedure is computationally expensive, making it difficult to interactively edit the transfer function and, therefore, reducing the options of interactivity when visualizing the volumetric data.

MORELAND and ANGEL [12] present a different solution to the pre-integration. They introduce the concept of partial pre-integration, where only part of the integral is computed and stored in a table. They modify the volume rendering integral, making it independent from the transfer function. A more detailed description on the construction of the ψ table of partial pre-integration can be found in the doctoral thesis of MORELAND [13].

This research work presents methods of direct and indirect volume rendering using some of the equations presented here. The partial pre-integration technique is also used, analyzing the advantages and disadvantages of different types of visualization and integration of the volume.

2.1.2 Visibility Ordering

Algorithms of direct volume rendering involve composition and, therefore, depend on a correct ordering of the cells for a given viewpoint to traverse the volume. Ray-casting algorithms employ an auxiliary data structure of adjacency so that when a ray leaves a cell it has enough information to find the next one. In contrast, cell-projection algorithms usually compute an approximate sorting in *object space*. Although there are algorithms for exact ordering of cells [14–16], they are complex and computationally expensive.

An approach aiming to combine the best of cell projection and ray casting is the *View-Independent Cell Projection* (VICP) of WEILER *et al.* [17]. Performing only ray casting inside each projected cell, VICP achieves high image quality consuming less memory than ray-casting algorithms. CALLAHAN *et al.* [18] present an approximate ordering approach named *Hardware-Assisted Visibility Sorting* (HAVS), which uses a hybrid direct volume rendering strategy similar to the VICP method. First, volume faces are ordered in object space by their centroids on the CPU and rendered using regular triangle rasterization. Afterwards, ray integration is evaluated in *image space*, with a refined ordering method using the *k-buffer* technique introduced in HAVS, where k determines the ordering precision, balancing performance and quality. One disadvantage is that by unifying sorting and rendering, the HAVS algorithm is limited to a fixed framework, preventing its use in an exact-ordering technique.

2.1.3 Ray Casting

The ray-casting technique dates from the 60's and its concepts were initially considered in volume rendering by BLINN [19] in the 80's. In Blinn's concept, a light ray is cast from each pixel on the screen, computing the absorption of light as it traverses the volume, as can be seen in the example of Figure 2.3.

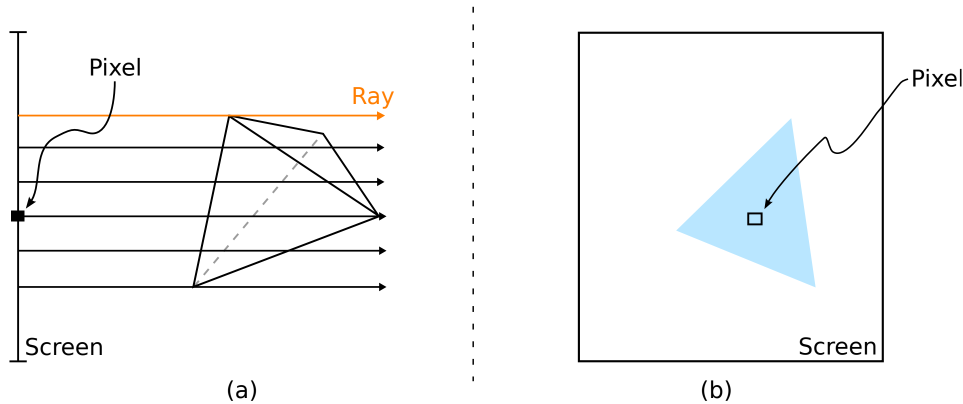


Figure 2.3: Example of the ray-casting technique. The side view of the rays traversing a tetrahedral cell of the volume per pixel of the screen is shown in (a) and the result in (b).

Later, the work of LEVOY [20] introduced a hybrid ray-casting algorithm rendering volumes using both polygons and the original data representation. His algorithm eliminates conversion artifacts, avoiding aliasing problems of other approaches. In a different perspective, the work of GARRITY [21] presented a more efficient approach to the ray-casting algorithm. His method casts rays in irregular datasets with transparency, using the connectivity between the volume cells to compute the path of the ray, i.e. the method finds all the cells intersected by the ray from one pixel to the end of the volume.

This approach was improved by BUNYK *et al.* [11] where all cells are broken into faces, in a pre-processing step. Thus, the *external faces*, also called *boundary faces*, are projected to determine the *visible faces* (see Figure 2.4). The visible faces define the entry point of the ray for each pixel of the image. With all faces created and stored in memory, BUNYK *et al.* [11] reduce the calculations related to the intersection and integration of rays improving in performance, but increasing the memory consumption. The work of Garrity and Bunyk *et al.* form the basis of the ray-casting algorithm presented in this thesis in Section 3.1.

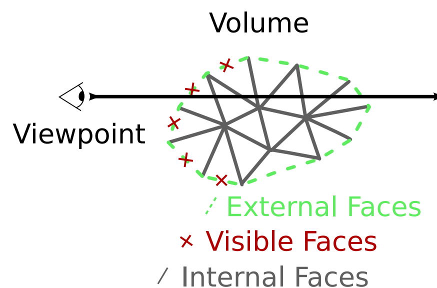


Figure 2.4: Types of faces in a volumetric dataset: external and visible faces are at the boundary of the volume; while internal faces are inside the volume.

WEILER *et al.* [22] present a method to perform ray casting on the GPU, using a fragment shader, called *Hardware-Assisted Ray Casting* (HARC). The HARC algorithm finds the entry point of the ray rendering the external faces, similar to the idea of Bunyk *et al.* The algorithm traverses the volume, storing the computation of the cells in textures.

ESPINHA and CELES [23] improve the HARC algorithm using partial pre-integration, rather than regular pre-integration, and employing a more efficient data structure than the original HARC implementation. Their proposed algorithm achieves high quality and allows the interactive editing of the transfer function.

In Chapter 5, the results of the HARC algorithm and the improved version of Espinha and Celes are analyzed and compared with the algorithms presented in this thesis.

2.1.4 Cell Projection

The cell-projection technique, also called *direct projection*, aims to generate images of the volume from the projection of its cells onto the screen. The projection is determined by projecting the three-dimensional cells to two-dimensional geometric primitives in the *image plane*, or *viewing plane*. After the projection of the cells is determined (see an example in Figure 2.5), the rasterization process fills the geometric primitives with fragments, which are combined to produce the final pixel.

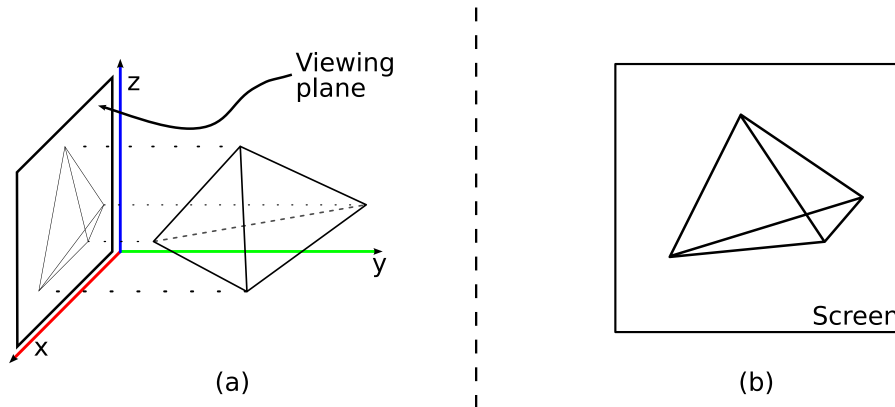


Figure 2.5: Example of the cell-projection technique. The projection of a volume cell is shown in (a) and the result in (b).

The main advantage of the cell projection over the ray casting is that, in the cell-projection technique, all ray intersections with the cell are computed implicitly in the projection. While, in the ray-casting technique, the intersection have to be computed for each ray, even if near rays intersect the same cell. The work of UPSON and KEELER [10] discusses the advantages and disadvantages of the ray-casting and cell-projection methods.

SHIRLEY and TUCHMAN [6] introduce the first cell-projection algorithm in irregular volumetric datasets. The algorithm deals exclusively with tetrahedral cells and, for this reason, it was named *Projected Tetrahedra* (PT). The PT algorithm consists of projecting and classifying the tetrahedra in the image plane and compositing them in visibility order. This algorithm forms the basis of the cell-projection algorithm presented in this thesis in Chapter 3 and it is explained in Appendix C.

KRAUS *et al.* [24] improve the quality of images generated by PT by applying a logarithmic scale for the pre-integration table. In addition, Kraus *et al.* point out the usage of high-precision textures (16 bits per color component) as responsible in part for the quality improvement.

WEILER *et al.* [17] developed another cell-projection method completely implemented on the GPU, using vertex and fragment shaders. Weiler *et al.*'s algorithm, called *View Independent Cell Projection* (VICP), applies the same vertex and fragment shaders independently of the viewpoint. The VICP algorithm combines the idea of ray casting, similar to HARC, with the cell-projection technique. Both algorithms of Weiler *et al.* (HARC and VICP) use a texture with the pre-integration table, as suggested by RÖTTGER *et al.* [8]. The final pixel color is determined by the composition process of the fragments using the Equations 2.4 and 2.5.

WYLIE *et al.* [9] implement the PT algorithm in the graphics card, using a vertex shader. The main problem in using the vertex shader for this algorithm is the fixed number of the input/output vertices, compromising the original idea of PT that defines a variable projection shape, with one to four triangles, depending on the projection of the tetrahedron. Aiming to solve this problem, the algorithm created by Wylie *et al.*, called *GPU-Accelerated Tetrahedra Renderer* (GATOR), classifies the projections of the tetrahedra in a different way from the PT algorithm of SHIRLEY and TUCHMAN [6]. GATOR uses a fixed topology, called *basis graph*, isomorphic to the 2D projection of the tetrahedron in the image plane. As a result, the limitation of the vertex shader with fixed number of input/output vertices is avoided, allowing GATOR to generate images like the PT algorithm.

The algorithm introduced by MARROQUIM *et al.* [25], called *Projected Tetrahedra with Partial Pre-Integration* (PTINT), performs the PT algorithm in two steps, avoiding the problem faced by GATOR. The PTINT algorithm, explained in Appendix D, forms the basis of the cell-projection algorithms presented in this thesis.

2.2 Mesh Processing

In this section we review techniques from the mesh processing area related to the algorithms presented in Chapter 4. This section is divided in two parts: in Subsection 2.2.1, the basic method of comparison by similarity using reflection is introduced; and in Subsection 2.2.2, similarity descriptors by signatures related to this thesis are discussed.

2.2.1 Reflective Symmetries

Symmetries in 3D models are normally detected and described as planar reflections among parts of a mesh [26–28]. One example of reflection plane can be seen in Figure 2.6, where the *Stanford Armadillo* model, of an armadillo puppet scanned at the Stanford University, presents a simple bilateral symmetry. The rendering of this model is part of this thesis and illustrates one of the models used in the tests of the method introduced here.

Figure 2.6: Example of reflection plane in the center of the Stanford Armadillo model, with each half presenting an approximate symmetric surface about the reflection plane.

In addition to these mirror-like symmetries, resemblances among details on a surface can also be found [29–31]. The concept of self-similarity of a meshed surface is defined by such resemblances, and naturally generalizes the concept of reflective symmetries. In this thesis, we say that two or more parts of a mesh are similar when they share local surface features, either reflective or not. Although the detection and structural analysis of reflective symmetries may be used to improve several

mesh processing tasks [32], there have only been a few papers [33, 34] dealing with repeated patterns but without the notion of mesh processing propagation introduced by this thesis.

Measurements of the self-similarity of meshes is employed as a tool in many applications, such as remeshing [35–37], rendering [38], shape matching [39], shape retrieval [40–42], and scene understanding [43]. These self-similarity detection techniques generally employ reflection as the base comparison method to determine similar regions. The use of reflection in the majority of these techniques comes from the fact that symmetries in nature tend to have repeated patterns between halves, such as a human face, an animal body, or a plant leaf. However, both natural and artificial objects may also contain more general classes of self-similarities. For instance, a computer keyboard has most of its keys sharing one identical shape.

One of the contributions of this thesis is a method to detect self-similarities not limited to the ones based on reflection, for scanned or modeled meshes, inspired by either natural or man-made objects.

2.2.2 Similarity Descriptors

One of the aspects which makes self-similarity identification especially challenging is the lack of a reasonable measurement tool for local shape comparison. GATZKE *et al.* [44] present a method to compare different local regions, introducing the *curvature map* of a point. They evaluate the mean and Gaussian curvature as a function of distance, using either neighborhood rings or geodesic fans as ZELINKA and GARLAND [33], for each point of the mesh. Thereafter, the self-similarity is measured as the difference among these curvature functions. The curvature map used by GATZKE *et al.* [44] works as a vertex descriptor, where similar vertices on the mesh have approximately equal descriptors.

Another example of vertex descriptor can be seen in Figure 2.7. The descriptor, or signature, in this example is a heightmap (shown in the bottom-left corner) considering as basis a region around the selected vertex (in red) and the tangent plane of the surface on this vertex defined by the vertex's position and normal. This vertex descriptor is part of the contribution of this thesis, which uses it instead of a curvature function as the signature of a vertex of the mesh to describe self-similarity. Both the construction of the heightmap and its usage are detailed in Chapter 4.

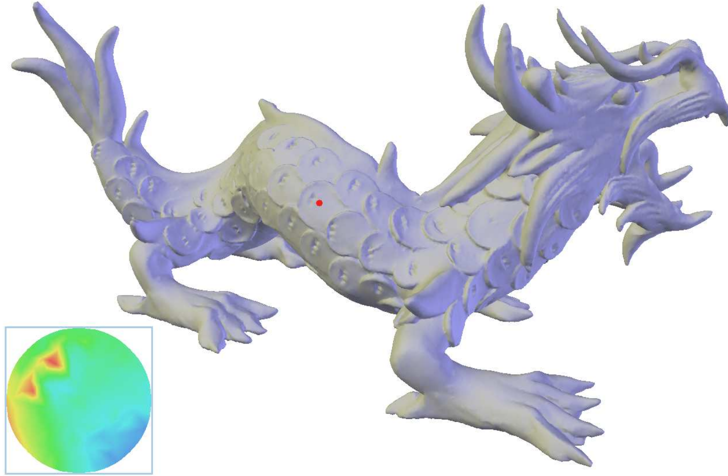


Figure 2.7: Example of the similarity signature of a vertex (in red) in the XYZ RGB Asian Dragon model. The signature is a heightmap of the region around the vertex. Note that the two cavities in the dragon scale are deeper than the other parts of the map in the bottom-left corner.

Techniques aiming to identify self-similarities mainly depend on the shape properties of a mesh. Much recent work in geometry processing considers the coordinate functions, or a more general function, as a signal defined on the meshed surface. One example is the method of spectral compression of meshes proposed by KARNI and GOTSMAN [45]. This method encodes geometry information as a compact linear combination of the orthogonal eigenvectors of the discrete graph Laplacian. The spectral compression method is based on the discrete differential geometry area not detailed in this thesis. The reader can refer to the complete course called *Discrete Differential Geometry: An Applied Introduction* [46] for more information.

VALLET and LÉVY [47]'s work describes how to use eigenvectors of a geometry-aware formulation of the Laplace-Beltrami operator as a function basis for mesh geometry representation; namely the *Manifold Harmonic Basis*.

OVSJANIKOV *et al.* [30] present a method to compute global intrinsic symmetries using eigenfunctions. Their method determines pose-invariant correspondence over the shape, i.e. symmetries that remain intact under isometric deformations. However, they restrict their method to reflective symmetries around the principal axes.

SUN *et al.* [31] improve this method by defining a local vertex signature, called the *Heat Kernel Signature* (HKS). The HKS is a multi-scale descriptor of the shape surrounding the vertex based on the temporal evolution of a heat-diffusion process on a mesh. Although this method provides a robust descriptor, it does not readily admit a region-based shape descriptor that we need in our approach.

After the identification of self-similarities on a mesh, the question that remains is how to succinctly describe these similarities. Methods defining signatures per vertex, such as the HKS, are one way to describe similarities. Another way is to use a more global data structure. SIMARI *et al.* [48] present an algorithm to compute the *folding tree*, a compact data structure using planar symmetry. Their algorithm, however, is restricted to find symmetry about principal axes. Symmetry detection about more general planes, such as those based on Principal Component Analysis (PCA), has been explored by KAZHDAN *et al.* [26] and CHENG *et al.* [49]. Perhaps the most general method for symmetry analysis of the entire object is the *planar-reflective symmetry transform* (PRST) [27] that captures reflective symmetries with respect to any plane. XU *et al.* [28] improve the PRST idea to allow for the detection of partial intrinsic rotational symmetries.

Similarity descriptors, ranging from local (per vertex) to global (the entire mesh), add information about a surface. GOLOVINSKIY *et al.* [32] present a framework to exploit such information, describing mesh processing tools to detect and preserve symmetries using the PRST and MITRA *et al.* [50]'s work on partial symmetry detection. The symmetry-aware mesh processing of Golovinskiy and colleagues is guided by the symmetries of a surface, while our method discovers more general similarities on the surface and allows for the processing to be carried out among similar regions.

The usage of a similarity descriptor to aid mesh processing is also explored by YOSHIZAWA *et al.* [51]. They use radial basis functions (RBFs) to approximate local vertex neighborhoods, and describe similarity by the difference among local shapes encoded in these RBFs. These differences are used as weights to remove noise from meshes based on non-local image denoising techniques. Another work aiming to filter noise from meshes is presented by SCHALL *et al.* [52]. In contrast with YOSHIZAWA *et al.* [51]'s work, they deal with range data, computing a point-wise height difference of the neighborhood rather than using RBFs. Our method also computes height differences around a vertex, however it can be used for any type of mesh, not only range image data. In addition, we show how our approach can be applied to several mesh processing techniques, such as parameterization and detail transfer.

Chapter 3

Volume Rendering

"... in 10 years, all rendering
will be volume rendering."
– Jim Kajiya at SIGGRAPH '91

In this chapter we present three cell-projection algorithms based on PT and one ray-casting algorithm, covering the contributions of this thesis in the area of volume rendering. The four algorithms are: *VF-Ray-GPU – Visible-Face Driven Ray Casting implemented on the GPU* – a memory-efficient ray-casting method fully implemented on the GPU; *RPTINT* and *IPTINT – Regular and Improved Projected Tetrahedra with Partial Pre-Integration* – two techniques based on the original PT algorithm [6], where the first is specialized to regular datasets and the second combines direct and indirect volume rendering; *HAPT – Hardware-Assisted Projected Tetrahedra* – a closer adaptation of the PT algorithm to graphics cards, where a flexible and efficient framework is presented, capable of extracting iso-surfaces and handling time-varying data on-the-fly. The volume rendering algorithms presented in this thesis cover cell projection and ray casting, handling both regular and irregular data, and employing both indirect and direct volume rendering techniques.

The VF-Ray-GPU, explained in Section 3.1, is based on the *VF-Ray* (*Visible-Face Driven Ray Casting*) [53] explained in Appendix B, that has been designed and implemented for GPUs by modifying its data structures. The RPTINT and IPTINT algorithms, presented in Sections 3.2 and 3.3, are based on the PTINT algorithm [25] which, in turn, is based on the PT algorithm [6], explained respectively in Appendices D and C. Finally in Section 3.4, the HAPT algorithm is detailed.

3.1 VF-Ray-GPU

VF-Ray The *Visible-Face Driven Ray Casting* (VF-Ray) algorithm is part of the ongoing work of Ribeiro's doctoral research, and it was published in the international conference *SIBGRAPI* 2007 [53]. In the VF-Ray algorithm, irregular meshes composed of tetrahedral or hexahedral cells are accepted as input. Each tetrahedral cell is composed of four faces and each hexahedral cell is composed of six faces. The algorithm is based on the following premise: the storing of the information about the faces of each cell is the key for memory consumption and execution time. This information is stored in a face data structure, that includes the geometry and the face parameters, constants for the plane equation defined by the face, which is the most consuming data structure in ray casting.

The basic idea behind VF-Ray is to explore ray coherence, improving caching performance by keeping in memory only the face data of the traversals of a set of nearby rays. The nearby rays, which will be cast through the neighboring pixels, are under the projection of a given visible face, as shown in Figure 3.1. This set of pixels is called *visible set*.

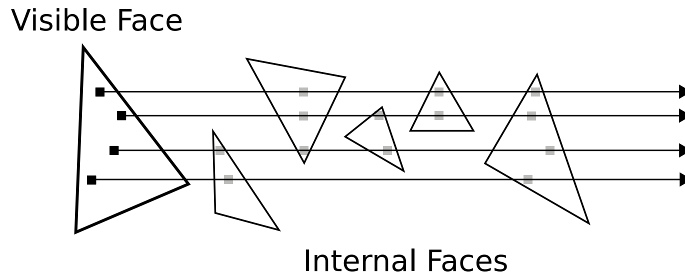


Figure 3.1: Ray coherence for one visible face.

The algorithm starts by determining the visible faces given the set of external faces of the volume and the current viewpoint. The external faces are pre-computed in a similar manner to the BUNYK *et al.* [11]'s algorithm, as well as the determination of the visible faces. In short, external faces only belong to one volume cell and are fixed for each volume, while visible faces are the external faces that have their normal vector pointing in the opposite direction of the viewing vector, which depends on the viewpoint.

The VF-Ray algorithm processes each visible face at a time, projecting it onto the screen and determining its visible set. For each pixel in this set, the ray traversal is done and each pair of intersections is computed. The internal faces are all the faces of the volume that are not external, that is, faces shared by two cells. The faces hit by one ray are determined testing the intersection of the ray against each face of each cell until the ray exits the volume. The faces intersected are created during the process, i.e. its interpolation parameters are computed.

The main idea of VF-Ray is to store each intersected face in a buffer, called *computedFaces*. Whenever a ray cast from the current visible set hits an internal face, already stored in the buffer, the VF-Ray algorithm just reads the face data from the *computedFaces* buffer, without having to recompute the face parameters. This interpolation parameters are used to compute the traversed distance l by the ray inside the cell and the entry and exit scalar values, s_f and s_b respectively, through the plane equation of the face and the line equation of the ray. Finally, the contribution of the cell to the color and opacity of the pixel is computed using l, s_f and s_b plugged to the integration model of Bunyk *et al.* explained in Section 2.1.1, where the difference of z front and back (z_f and z_b) given by Δ_z is equivalent to the distance l in a not necessarily orthogonal projection.

VF-Ray algorithm explores ray coherence, using the visible face information to guide the creation and destruction of face data in memory. The faces intersected by neighbor rays tend to be the same, as shown in Figure 3.1, and their data are kept in the *computedFaces* buffer while the visible set is being computed. When all pixels in a visible set are processed, the VF-Ray algorithm clears the *computedFaces* buffer and proceeds to the next visible face.

Handling Degeneracies Degenerate situations occur when a ray hits an edge or a vertex of the volume. The method and the data structures proposed by Bunyk *et al.* do not correctly deal with these degenerate cases, generating incorrect pixels colors. In the VF-Ray algorithm, on the other hand, theses cases are handled in a manner similar to that proposed by PINA *et al.* [54]. The idea is to pre-compute a new data structure called *Use_set*, containing all the cells incident to each vertex of the volume. In this way, the *Use_set* can be used to continue the ray traversal, even if the ray hits a vertex or an edge, ensuring that the final image will be correctly generated.

VF-Ray-GPU The algorithm *Visible-Face Driven Ray Casting implemented on the GPU* (VF-Ray-GPU) relies on the GPGPU (*General Purpose Computation on Graphics Hardware*) implementation to parallelize the original VF-Ray algorithm. In this approach, the graphics hardware is assumed to be a CPU coprocessor capable of performing high arithmetic-intensity tasks, regardless of the GPU-specific computations. The VF-Ray-GPU algorithm was published in the international symposium *Volume Graphics* 2008 [55].

The architecture used to implement the ray-casting algorithm presented in this thesis – VF-Ray-GPU – was CUDA [4] due to its simplicity, portability, and efficiency for GPUs. As explained in Section 1.3, the implementation in CUDA is done through a kernel associated with a grid of blocks, where each block has multiple threads running in parallel. The threads have a limited implementation capability, which results in two main problems: first, if the function associated with the kernel is too extensive, the execution may fail; and second, the amount of allocated resources, e.g. registers, may be too large, making it impossible to compile the code. This limitation of resources depends on the number of threads per block, since the higher this number, the lesser resources are available per multiprocessor.

The VF-Ray-GPU algorithm is divided into three steps and, as a result, three different kernels (see Figure 3.2). In the first kernel, the external faces are read and the visible faces are determined. The second kernel computes the projection of each visible face, splitting them in pixels on the screen space. Finally, the third kernel evaluates the ray-casting algorithm over each visible face pixel previously computed.

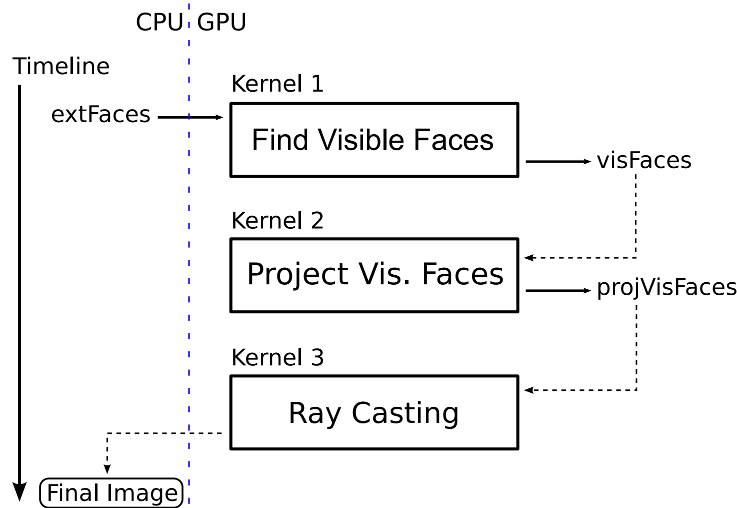


Figure 3.2: The three kernels of the VF-Ray-GPU algorithm. The external faces – *extFaces* – are read by the kernel 1 (*Find Visible Faces*), responsible for determining the visible faces – *visFaces*. Then, the visible faces are projected onto the image plane by the kernel 2 (*Project Vis. Faces*) and the ray casting is carried out using the projected faces – *projVisFaces* – by the kernel 3 (*Ray Casting*).

Data Structures Before processing the three GPU kernels, the following data structures are used by the VF-Ray-GPU algorithm. The tetrahedra connectivity (*conTet*) is computed similarly to the work of GARRITY [21] and BUNYK *et al.* [11], i.e. from the basic information of the volumetric data: the list of vertices (*vertList*) and the list of tetrahedra (*tetList*). The *conTet* list stores for each face of each tetrahedron, the tetrahedron index t_i that shares that face. In the case of an external

face, the t_i index stored is the tetrahedron index of the cell itself. In Figure 3.3 the data structures used by the VF-Ray-GPU algorithm are shown, illustrating the first tetrahedron ($tetrahedron_0$). The *vertList* contains the coordinates x, y and z, and the scalar value s of each vertex. The *tetList* contains the vertices indices v_i that defines each tetrahedron.

To avoid building other data structures, the GPU algorithm uses the vertex order inside the *tetList* to determine each tetrahedron face. Thus, the vertices of the face f_i are v_i, $v_{(i+1)mod4}$ and $v_{(i+2)mod4}$. For example, the face f_2 of some tetrahedron t_i is composed by the vertices v_2, v_3 and v_0 of t_i, as can be seen in Figure 3.3. In addition to these data structures, the VF-Ray-GPU algorithm uses the external faces list (*extFaces*) pre-computed in the same way as the original VF-Ray algorithm.

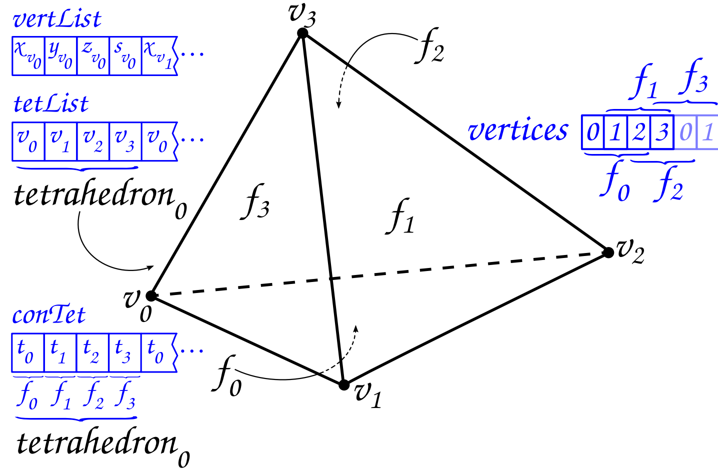


Figure 3.3: The basic data structures of the VF-Ray-GPU algorithm.

The original VF-Ray algorithm stores three vertex indices inside its face data structure, pointing to the three vertices of the face in the *vertList*. In the GPU algorithm only two indices are stored for each face created. These indices are the tetrahedron t_i and the face f_i, which are sufficient to identify the vertices of each face, as explained above. This reduction in the number of indices used in identifying a face saves memory, but increases the amount of accesses since the *tetList* must be accessed prior to reading each vertex of the face.

First Kernel The first kernel reads the external faces from texture memory and computes the parameters, i.e. plane equation coefficients, for each one. This computation is called face creation and involves solving two 3×3 linear systems: one to interpolate the z coordinate; and another to interpolate the scalar value s. In this kernel, only the z coordinate interpolation is done for each external face and, once this face is checked as visible, the s value interpolation is done. Both interpolation parameters are stored in the list of visible faces (*visFaces*).

The visibility for an external face is tested by comparing the z coordinate of the fourth vertex of the tetrahedron, i.e. the vertex that does not belong to the face, with the z coordinate of its projection on the external face. The fourth vertex projection is performed using the z coordinate interpolation parameters computed for the external face. The face is visible if the projected z is smaller than the z coordinate of the fourth vertex. Note that this comparison is equivalent to the *back face culling* test, where the face normal is compared against the viewing vector using dot product.

Only the visible face parameters are written in CUDA global memory, that is, the z and s interpolation parameters are stored in the *visFaces* list, as illustrated in Figure 3.2.

The first kernel employs the face creation and the visible face test, i.e. back face culling, for each thread, using the maximum number of threads available per block. The number of external faces is fixed for each volume data and thus is the number of threads across all blocks in this kernel. The computation time and memory footprint spent in the first kernel corresponds to less than 5% of the total (time and memory). Nevertheless, the main role of this kernel is to reduce the number of threads that will be used by the next two kernels. From now on, they will run over the *visFaces* list instead of the *extFaces* list. In this way, the number of threads in the kernels 2 and 3 are equal to the number of visible faces rather than the number of external faces, reducing the number of threads by almost 50%.

Second Kernel The second kernel reads the coordinates of the vertices of each visible face from the *vertList* in texture memory. The vertex indices to access the *vertList* are read from the *tetList* in global memory. Note that texture components can not be indexed directly, forcing the implementation of the VF-Ray-GPU algorithm to store the *tetList* in global memory, instead of texture memory, in order to avoid unnecessary branches.

The vertex coordinates are used to project the visible faces onto the screen space. The bounding box of the projection is computed and stored in the projected visible faces list (*projVisFaces*), as illustrated in Figure 3.2, which is written in global memory to be used by the next kernel.

The second kernel uses one thread for each visible face computation. As the first one, this kernel is performed by each thread using few resources of a block. Therefore, the number of threads per block can be maximized and the computation time and memory footprint are unnoticeable.

Third Kernel The third kernel reads the *visFaces* list, computed in the first kernel, and the *projVisFaces* list, computed in the second kernel. The visible face pixels, which define the visible set, are determined using the bounding box of the projection and computing a point-inside-triangle test. This test employs simple cross-product operations. For each visible face pixel determined, the ray casting is triggered using the visible face as the first entry face. All the pixels will use the face parameters stored in the *visFaces* list. From this point, the ray path is discovered by computing the intersections of the ray with each internal face.

Similarly to the original VF-Ray algorithm, the internal faces are stored in a buffer called *computedFaces*. However, the GPU algorithm is designed to run in parallel taking advantage of the ray coherence inside each thread computation. In order to avoid dynamic allocation, the *computedFaces* buffer is allocated in CUDA local memory for each thread with a fixed size and it is indexed by a hash table. The hash index is the z coordinate centroid of the face, the idea can be seen in Figure 3.4.

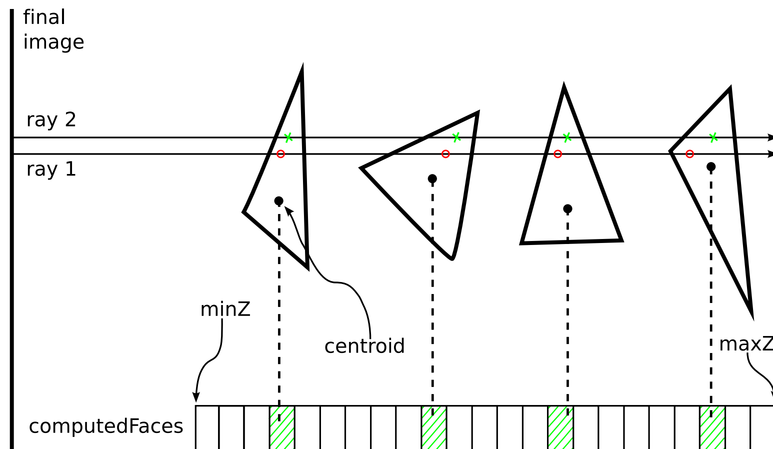


Figure 3.4: The *computedFaces* buffer is used to store the faces previously created (red circles) in order to be read in the future (green crosses). The *centroid* is used to hash the faces in the buffer.

To perform the hash of the faces, the first buffer position is associated with the minimum z coordinate ($minZ$) of the volume dataset, while the last position is associated with the maximum z coordinate ($maxZ$). Figure 3.4 presents two nearby rays processed by the same thread one after the other. Ray 1 creates four internal faces and ray 2 reads them from the *computedFaces* buffer. This buffer is used to store the face parameters and two indices: the tetrahedron id t_i and the face id f_i. These two indices ensure that the face to be read is the face that was hit, in case of collision in the hash slot. In contrast, the original VF-Ray algorithm spends more memory storing four face indices for each tetrahedron to point to the *computedFaces* buffer.

The third kernel divides the CUDA grid in blocks (see Figure 3.5) associating one block to one visible face. Each block is, in turn, divided into threads where each one computes a small set of pixels. Unlike the first two kernels, the amount of computation performed in the ray traversal consumes much more resources of a block, making the computation of all visible face pixels by one thread impractical.

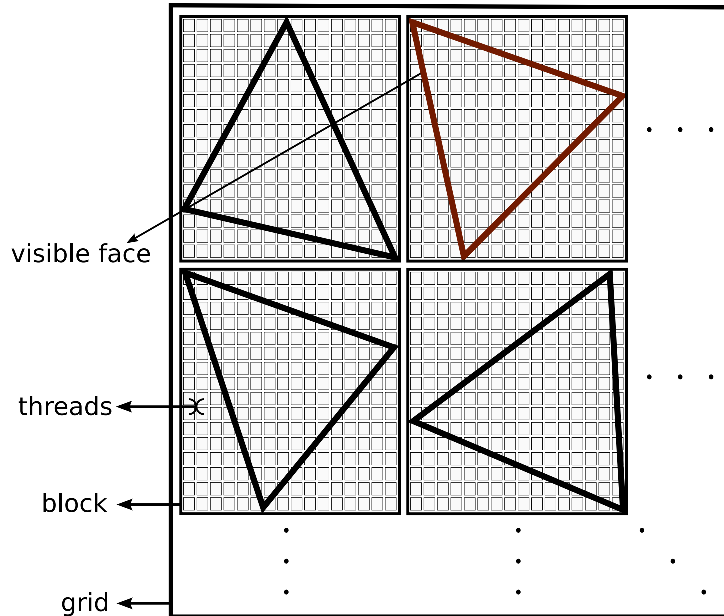


Figure 3.5: The grid/blocks/threads scheme used by the third kernel of the implementation of the VF-Ray-GPU algorithm. Each block inside the grid computes one visible face, and each thread inside the block computes a small set of pixels. The number of pixels in this set depends on the number of pixels of the visible face, balancing the number of threads per block with the size of the set of pixels treated by each thread.

Handling Degeneracies The degenerate cases are handled differently than the original VF-Ray algorithm. While in VF-Ray the *Use_set* list is pre-computed and used every time a ray hits a vertex or an edge; in the GPU algorithm, the ray is perturbed to prevent the degenerate cases, continuing the next iteration with the ray in the same direction without considering the displacement. The results obtained by this new approach are similar to the original VF-Ray, but save the memory spent to keep the *Use_set* list.

Source Code The VF-Ray code for CPU and GPU using C/C++ and C for CUDA is made available with this thesis at:

http://code.google.com/p/vfray.

3.2 RPTINT

The *Regular Projected Tetrahedra with Partial Pre-Integration* (RPTINT) algorithm was published in the international conference *GRAPP* 2007 [56], and it is the first extension of the PTINT algorithm presented in this thesis. The *Projected Tetrahedra with Partial Pre-Integration* (PTINT) algorithm was the main theme of the master of science dissertation done by me in 2006 [57], and it was published in the SIBGRAPI of the same year [25].

The basic idea in this first extension is to use the PT algorithm [6] on the GPU taking advantage of the volume data regularity. A regular grid consists of voxels (volume elements) regularly spaced where the topology is implicit. Regular data cells are hexahedra, where each vertex of the hexahedron corresponds to a voxel with its scalar value assigned.

The algorithm consists of four steps, the first three steps take place on the CPU, whereas the last is performed by the GPU (see Figure 3.6). First, the projection of a single hexahedron is determined by splitting it into five tetrahedra. Second, a traversal order for the whole volume is determined in constant time. The remaining step on the CPU consists of allocating the volume information in a Vertex Array structure. Lastly, triangle fans corresponding to the projected tetrahedra are rendered and composited in a back-to-front order using the GPU.

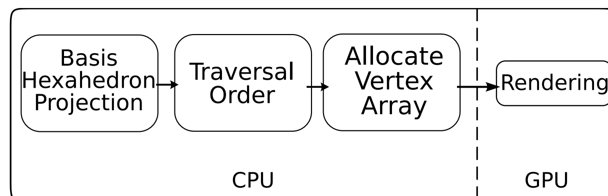


Figure 3.6: Overview of the RPTINT algorithm. In the first step the basis hexahedron is projected, dividing it into five tetrahedra and using the original PT algorithm. Afterwards, the rendering order of the volume cells is determined (*traversal order*). In the third and last step on the CPU the vertex array is allocated and it is used by the fourth step on the GPU responsible to render the volume.

Basis Hexahedron Projection Since the PT algorithm requires a tetrahedral mesh as input, the regular volume data must be preprocessed by subdividing each hexahedron into five tetrahedra. We term each such set of tetrahedra a *volume unit* or *volunit* for short. A key observation is that, by rendering the volume in orthographic projection, all volunits are projected to the screen in exactly the same way. Thus, to avoid redundant computation, the projection parameters are computed only once for a *basis hexahedron*. The PT algorithm is used in the projection of the volunit related to the basis hexahedron.

Each *basis tetrahedron*, i.e. tetrahedron resulting from the subdivision of the *basis hexahedron*, is projected to screen coordinates and its projection class is determined by means of four cross-product tests of the PTINT algorithm (explained in Appendix D). After projection, the *thick* vertex, defined as the entry point of the tetrahedron where the ray traverses the maximum distance l, and the front s_f and back s_b scalar values can be computed with at the most two segment intersections (depending on the class).

For each of the five *basis tetrahedra*, the following projection values are computed and stored:

- *basis projection class*;
- coordinates of the *basis vertices* projected;
- coordinates of the *basis thick vertex*;
- *basis intersection parameters* for computing the front and back scalar values;
- *basis rendering order*.

Rendering As the original PTINT algorithm, the rendering of the volume by RPTINT is performed by sending each tetrahedron as a triangle fan to the graphics card. The primitives are drawn in a back-to-front order and the resulting pixel fragments are composed using a blend function. The *basis hexahedron* is iteratively displaced to the position of each volunit, and the stored *basis vertices* are used to compose the tetrahedra triangles. Each triangle fan is rendered with the number of triangles relative to its *basis projection class* following the *basis rendering order*. The *basis thick vertex* is used as the first vertex of the fan, i.e. the center vertex of the triangle fan.

Even though the geometry can be resolved by just displacing the *basis hexahedron*, the colors and opacity values are unique for each vertex and must be computed on-the-fly for each volunit. In fact, the final color is computed only in the fragment shader in the final step on the GPU. The values assigned as the vertex color are its front s_f and back s_b scalar values and its thickness l.

For each *thin* vertex, all others vertices excluding the thick vertex, the scalar front and back values are the same (the original scalar of the volume data). Furthermore, their thickness value is always zero since the ray traverses no distance inside the tetrahedron at these vertices. On the other hand, the scalar values for the thick vertex are calculated using the *basis intersection parameters*, while the *basis thickness* was already computed for each *basis tetrahedron* and it does not change among the volunits of the data. The definitions for thick and thin vertices are given in the explanation of the PT algorithm in Appendix C.

Vertex and Fragment Shaders The vertex coordinates are computed by RPTINT on-the-fly using vertex shader and based on the three integer indices of the vertex in the lattice (i, j and k of the 3D grid of the regular data are used to position each vertex of the volume). This computation requires several parameters previously determined on the CPU for the *basis hexahedron* and passed to the shader as global values, called *uniform variables* in GLSL [2].

Each grid vertex is sent to the GPU multiple times, one for each incident tetrahedron. Thus, along with the vertex grid coordinates, additional information is coded in the vertex normal, namely the tet_{id} (local tetrahedron index) and the $vert_{id}$ (local vertex index). The tet_{id} is stored in the x normal coordinate and identifies which of the five tetrahedra (inside the hexahedron) is being rendered. The $vert_{id}$ is stored in the y normal coordinate and identifies the vertex inside the tetrahedron. It should be noted that this coding scheme shifts most of the computational load from the CPU to the GPU.

The fragment shader receives trilinearly interpolated vertex colors (s_f, s_b, l) for each triangle fragment. The interpolated scalar values are used to lookup the chromaticity and opacity values in a transfer function table. This table is stored in a 1D texture in the same way as the original PTINT algorithm. Additionally, aiming to improve the algorithm performance, the tetrahedra with all vertices defined with zero opacity (by the editing of the transfer function) are discarded, that is, they are not sent down the algorithm's pipeline.

Figure 3.7 summarizes the algorithm's pipeline on the GPU (step 4 – rendering). Each volume hexahedron is sent to the GPU as five triangle fans, where each fan corresponds to one projected tetrahedron. The vertex shader uses the *basis hexahedron* information, stored globally as uniform variables, to correctly displace each vertex. The rasterization process is responsible for interpolating the s_f, s_b and l values inside each triangle.

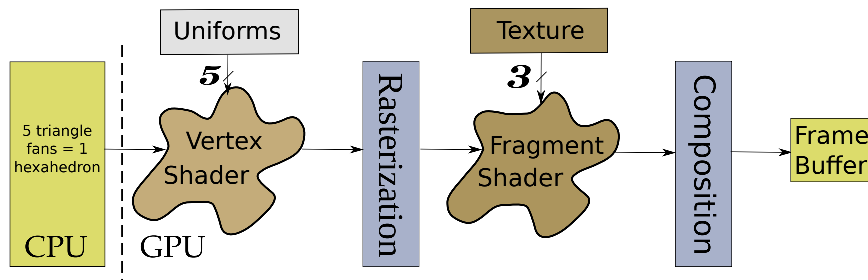


Figure 3.7: Pipeline of the RPTINT algorithm. The five projected *basis vertices*, including the thick vertex, are read in the vertex shader. While the fragment shader reads three textures: one table with results of the exponential function $f(x) = e^x$ to speed up the evaluation; one table with the transfer function; and the ψ table of partial pre-integration to improve the quality of the original integration of PT.

In the fragment shader, the interpolated values s_f, s_b and l are used to compute the final color of the fragment. The front and back scalar values are transformed in colors by the transfer function. The exponential function is used as a lookup table to calculate the opacity of the fragment. The front and back colors are employed in the lookup of the ψ table of partial pre-integration by MORELAND and ANGEL [12], in the same way as the original PTINT algorithm. The final color and opacity of the fragment, the output of the shader, are composed in the last stage of the pipeline (see Figure 3.7) before forming the final image in the screen.

Data Structures In the RPTINT algorithm the vertices and tetrahedra textures are not stored. These were responsible for representing the volume on the GPU in the original PTINT algorithm. In RPTINT, the tetrahedra projection and classification are performed on the CPU (in the first step) instead of the GPU as PTINT. With this, the memory consumption on the GPU is extremely reduced, being constant and independent of the size of the volume rendered.

The volume is described through the arrays sent to the GPU in the third step of the algorithm. The three arrays are: *vertex*, *color*, and *normal*. Each array has a fixed size and contains the five vertices of each one of the five tetrahedra (four vertices plus the thick vertex). Figure 3.8 shows the data structures of RPTINT with the part associated with one example of projection in detail.

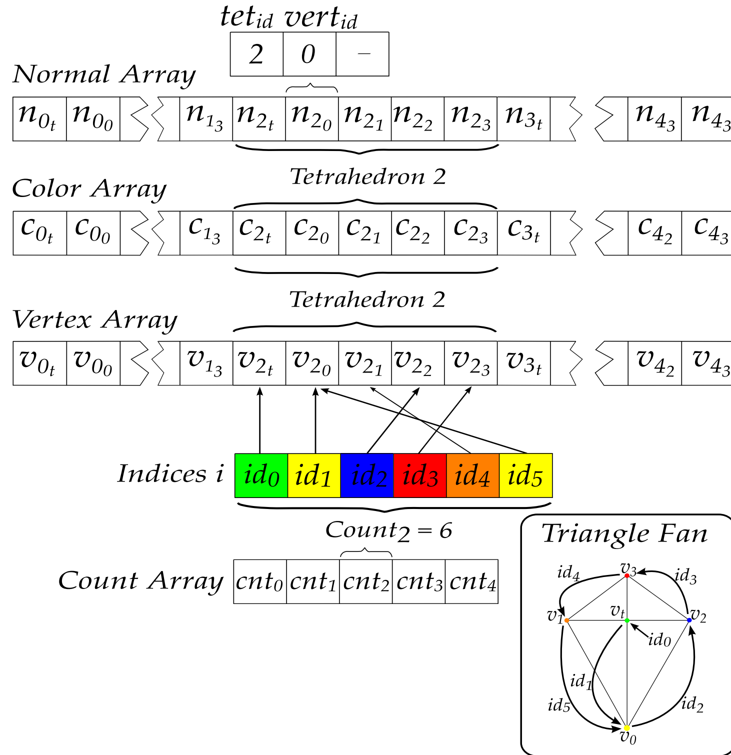


Figure 3.8: The vertex, color and normal arrays form the data structures responsible to send the volume information from the CPU to the GPU in the RPTINT algorithm.

Additionally, a count array is used to determine the order and quantity of the vertices in the triangle fan, as illustrated in Figure 3.8. The case of the tetrahedron shown in Figure 3.8 (bottom-right corner of the figure) is of projection class 2, generating four triangles in the fan to render. The size and structure of the arrays used by RPTINT do not change with the viewpoint, since they are built and stored during pre-computation. The third step of the algorithm is only responsible for updating certain values, which depend on the shape of the projection, and for sending the stored arrays.

Cell Sorting One disadvantage of cell-projection algorithms is the necessity of sorting. Sorting millions of cells is computationally expensive, requiring auxiliary data structures and pre-processing steps [14–16, 58]. However, the RPTINT algorithm avoids the cost of sorting the cells benefiting from the fact that the volumetric data is regular and hence implicitly sorted as a 3D array. The only step required is to determine a traversal rule for this data (second step on the CPU), which can be done in constant and negligible time. In fact, only 8 possible traversal orders are sufficient to render the volume from any possible angle.

Let $\vec{v} = \{v_x, v_y, v_z\}$ be the viewing vector, i.e. a vector pointing to the observer. Then, a positive v_x indicates that the *volunits* must be traversed in ascending order of x indices, whereas a negative v_x would indicate a descending order. A similar rationale may be used for the other axes. Note that the relative order in which the axes are traversed does not matter for regular data, that is, iterating in the order x, y and z will produce the same result as iterating in z, y and x or any other permutation.

Interaction with the Volume The interaction with the volume data is improved in the RPTINT algorithm adding a clipping tool, besides the interactive transfer function editing tool that appears in the original PTINT [57]. The clipping tool acts directly on the volume image. By selecting rectangular areas, the volume can be trimmed and smaller features enlarged. The only limitation is that the clipping must be parallel to one of the volume's bounding box sides to ensure that the regular properties are maintained.

Source Code The RPTINT code, using C/C++ and GLSL, is made available with this thesis at:

http://code.google.com/p/rptint.

3.3 IPTINT

The *Improved Projected Tetrahedra with Partial Pre-Integration* (IPTINT) algorithm was published in the international journal *Computer Graphics Forum* in 2008 [59], and it is the second extension of the PTINT algorithm presented in this thesis. In the published paper, the original PTINT algorithm is the variant of PT on the GPU with partial pre-integration, and the IPTINT algorithm corresponds to the variant with both rendering techniques: iso-surfaces and partial pre-integration.

The IPTINT algorithm uses the same idea of the PTINT algorithm, performing volume rendering almost entirely on the graphics card and dividing the execution in two main steps in fragment shaders. In the first step, all relevant information of each tetrahedron is computed, that is, the projection class, the thick vertex properties, and the z coordinate of the tetrahedron's centroid. In the second step, the vertices' scalar and the gradient vectors (corresponding to the scalar field variation) are interpolated in the rasterization step and used to compute the chromaticity and opacity values of the fragment. The interpolated scalar is used to detect iso-surfaces, while the interpolated gradient is used to illuminate the iso-surface.

Gradient Information Unlike the original PTINT algorithm, the thick vertex has the front (g_f) and back (g_b) gradient information. These values are computed in IPTINT in the same way that the scalar front (s_f) and back (s_b) values are computed in PTINT, by interpolating the gradients at the vertices. To compute these values, a third texture called *Gradients Texture* is employed. Each *RGB* texel of this texture stores the x, y and z coordinates of the gradient vector pre-computed per vertex (see Figure 3.9).

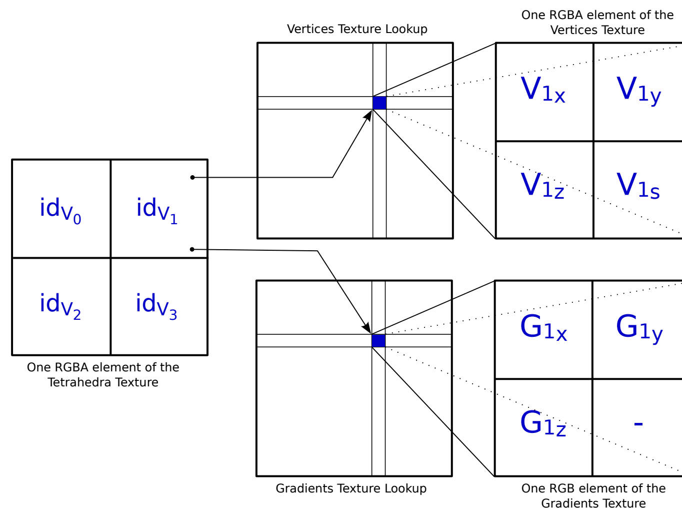


Figure 3.9: Lookup of the vertex data on the first fragment shader. Each texel of the Tetrahedra Texture contains the indices of the corresponding tetrahedron's four vertices in the Vertices and Gradients Textures.

Aside from reading the gradients and interpolating them, the first step of the IPTINT algorithm is identical to the original PTINT. The I/O of the first step is changed to support the new gradient information, as can be seen in Figure 3.10. Note that, for the IPTINT approach it is necessary to read four textures and render four different framebuffers. This rendering is done using a technique called *multiple render targets* (MRT). In the case of PTINT, just three textures and two framebuffers are used (see Appendix D). With this, two gradient vectors per thick vertex of each tetrahedron are read back to the CPU.

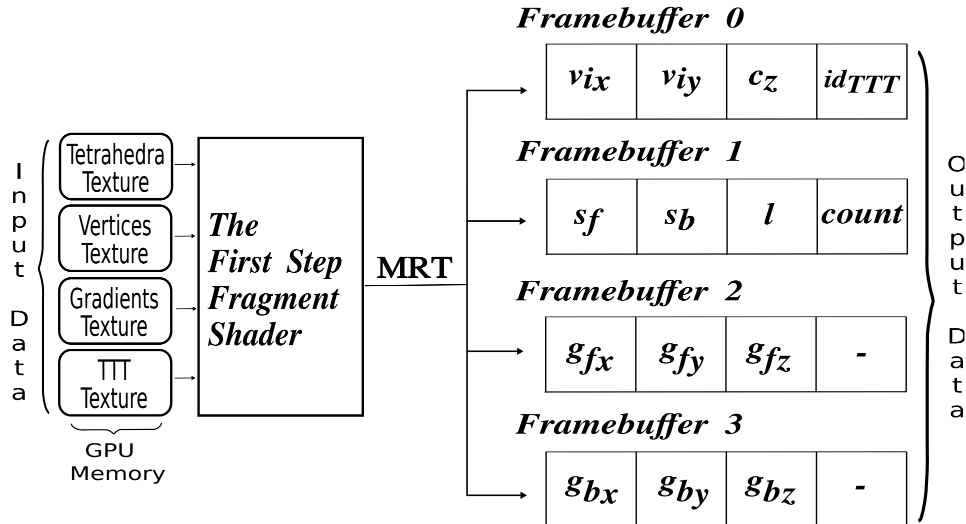


Figure 3.10: I/O scheme of the first step of the IPTINT algorithm. The Gradients Texture is used to read the gradient vector of each vertex in the fragment shader. The framebuffers 2 and 3 are used to pass the information of the computed front and back gradient vectors from the first to the second step of the algorithm.

Preparing the Arrays for Rendering The intermediate step between the first and second step on the GPU is performed by the CPU, and consists of: ordering the cells; and organizing the arrays for rendering. In the same way as the original PTINT, IPTINT uses two approaches to the ordering of cells: a simple and inaccurate ordering by slices (using *bucket sort*) for when the volume is being rotated; and a default and more costly ordering (using *merge sort*) for the first still frame of the volume. The sorting by slices divides the volume in intervals of distance from the viewpoint and the intervals, or slices, are sorted leaving the tetrahedra inside each interval without sorting. The merge-sort approach sorts all the tetrahedra of the volume. Both sortings use the centroid z of the tetrahedra (c_z) computed in the first step of the algorithm. The usage of the tetrahedron's centroid is an approximate way to position it in space for sorting.

When organizing the arrays for rendering, the IPTINT algorithm employs four data structures rather than two as the original PTINT. These four structures store the coordinates, color values (assigning s_f, s_b and l to them) and gradient vectors to all vertices of the volume (see Figure 3.11).

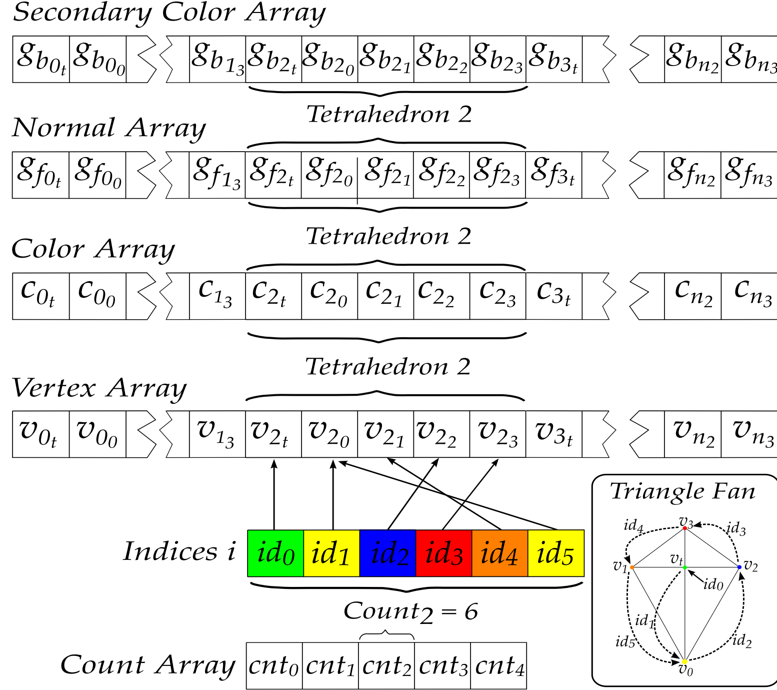


Figure 3.11: The arrays of vertex, color and gradients (as normal and secondary color) form the data structures responsible to send the volume information from the CPU to the GPU in the IPTINT algorithm.

The data structure in form of four arrays contains the tetrahedra grouped in five elements: the thick vertex and the four original vertices of the tetrahedron. Since changings in the viewpoint only updates some of the data, such as position, color and gradient of the thick vertex, most of these arrays are constant. This allows OpenGL [1] to keep most of the volume information in GPU memory, avoiding CPU–GPU data-transfer overhead.

The vertex array contains the $\{x, y, z\}$ coordinates of each vertex. The color array contains the values $\{s_f, s_b, l\}$ rather than actual colors, which will be computed in the fragment shader of the second step. Finally, the $\{x, y, z\}$ vectors of the front and back gradients are stored in the normal and secondary color arrays, respectively. Note that, for a given thin vertex v_i, $s_f = s_b = s_i$, where s_i is the scalar value of the vertex v_i, the thickness $l = 0$ and $g_f = g_b = g_i$, where the g_i is the gradient vector of the vertex v_i.

The four arrays are rendered using the OpenGL function *glMultiDrawElements*, as done by the PTINT and RPTINT algorithms. In addition to the four arrays with the volume information, there are the indices and count arrays used to guide the rendering by the *glMultiDrawElements* function. The indices array is divided into n groups, where n is the number of tetrahedra. For each tetrahedron, the correct order to render the triangle fan is stored as six integers (highlighted group called *Indices i* in Figure 3.11) that index the vertices of the tetrahedron. The count array contains the number of vertices in each fan. It is important to remember that the maximum number of vertices in a fan is six (cases of projection class 2) resulting in four triangles. For cases with less than six vertices, the indices array is accessed only up to the position cnt_i.

Rendering Iso-surfaces The second step of the IPTINT algorithm is responsible for computing the color of each fragment. As the original algorithm, IPTINT uses the values interpolated by the graphics card's rasterizer in the computation of color and opacity to generate the final image. However, the interpolated values in each fragment are not only $\{s_f, s_b, l\}$, but also $\{g_{f_x}, g_{f_y}, g_{f_z}\}$ and $\{g_{b_x}, g_{b_y}, g_{b_z}\}$, that is, the attributed values of each vertex (color, normal, and secondary color).

Apart from volume rendering, the IPTINT algorithm also allows the rendering of iso-surfaces. An iso-surface can be defined by a function of binary segmentation $f(s)$ applied to the volume data, where $f(s)$ returns 1 if the value s is considered part of the surface and 0 if not. When $f(s)$ is a step function, $f(s) = 1, \forall s = s_{iso}$, where s_{iso} is called *iso-value*, the resulting region is an *iso-surface*. For the case of an interval $[s_1, s_2]$ in which $f(s) = 1, \forall s \in [s_1, s_2]$, where $[s_1, s_2]$ is called *iso-interval*, the resulting region is a structure called *level set*.

An iso-value is associated with a scalar value and, for each fragment, IPTINT determines if the iso-surface crosses the corresponding tetrahedron or not. Since each fragment contains the interpolated front and back scalar values, s_f and s_b respectively, the evaluation consists of testing if the iso-value is inside this interval. If not, the fragment is computed using only the partial pre-integration technique, that is, the direct volume rendering technique of the original PTINT. On the flip side, if the iso-value is between s_f and s_b, the iso-surface crosses the corresponding tetrahedron and the color computation of that fragment is performed considering the Phong illumination model [60] and using the interpolated gradient as the surface normal. Figure 3.12 shows an example of one iso-surface found inside a tetrahedron and the resulting fragments inside the triangle fan of that tetrahedron (example of projection class 1 generating three triangles).

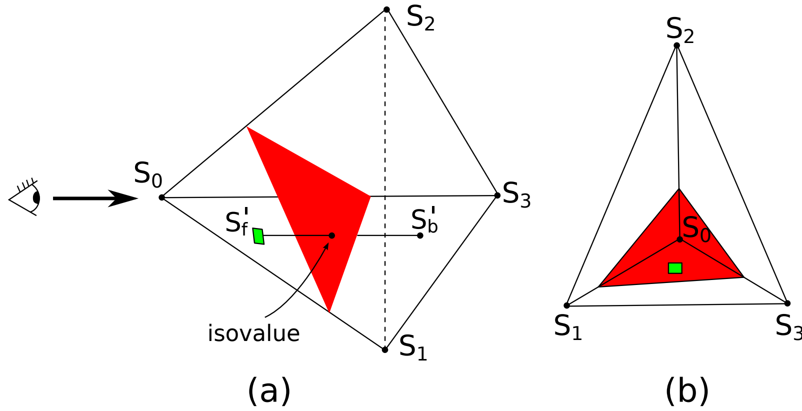


Figure 3.12: Example of an iso-surface crossing a tetrahedron (a) and the projected tetrahedron with a fragment (in green) rendered inside the iso-surface (b). Both direct and indirect volume rendering techniques are used in the final color evaluation of the highlighted fragment.

This approach is similar to the work of KLEIN *et al.* [61]. However, in the IPTINT algorithm the iso-surface is not computed explicitly, but by fragment, taking advantage of the rendering pipeline of the basis PTINT algorithm. With this advantage, there is no need to deal with different cases where the iso-surface crosses the tetrahedron, as in the algorithm of PASCUCCI [62].

The iso-surfaces are rendered using the illumination model of Phong. The interpolated gradients act as approximate normal vectors of the surface in the computation of diffuse and specular lighting. The front and back gradients are linearly combined using a weight value relative to the distance of the iso-surface to the ray's entry and exit point inside the tetrahedron. This means that, if the iso-surface is closer to the ray's exit point, for instance, the back gradient will have a greater weight in the evaluation of the surface normal, than the front gradient.

This technique allows a hybrid visualization of the volume (direct and indirect) with multiple iso-surfaces. The iso-surfaces can be rendered completely opaque (making the visualization strictly indirect) or with transparency. And, since all iso-surface computation is done during the second fragment shader, the IPTINT algorithm does not need to employ an additional step to extract iso-surface on the CPU or the GPU.

Source Code The PTINT and IPTINT code with the *Ternary Truth Table* (TTT), using C/C++ and GLSL, is made available with this thesis at:

http://code.google.com/p/ptint.

3.4 HAPT

The *Hardware-Assisted Projected Tetrahedra* (HAPT) algorithm was recently accepted in the international symposium *EuroVis* 2010, to be published in a special issue of the *Computer Graphics Forum* journal [63].

The HAPT algorithm is based on the PT algorithm, explained in Appendix C, and it was developed to completely explore the graphics card's resources. Unlike PTINT, and the two extensions presented previously in this thesis (RPTINT and IPTINT), HAPT uses the geometry shader, in addition to the vertex and fragment shaders, adapting the projected tetrahedra idea in a more efficient and closer way to the original PT algorithm. The problem of cell sorting is handled by HAPT using a fast sorting algorithm, *quicksort* on the GPU by CEDERMAN and TSIGAS [64], adapted to our scenario and implemented in CUDA [4]. Once the cells are sorted, HAPT performs the PT algorithm in a single GPU step, while taking full advantage of the three types of shaders and the processors dedicated to triangle rasterization. In this way, HAPT has four main features: first, it performs volume rendering of irregular structures in a single rendering pass after sorting; second, it consumes almost no GPU memory since the tetrahedra are streamed to the graphics card; third, in addition to direct volume rendering, iso-surfaces can be extracted and rendered on-the-fly, and time-varying data are also easily handled; and finally, HAPT's framework allows for easy exchange of its modules, such as sorting, outside the rendering pipeline, or volume integration methods, inside the pipeline, providing greater implementation flexibility.

Algorithm's Framework HAPT's framework is presented in Figure 3.13. First the visibility order is computed using any CPU or GPU method. The ordered tetrahedra are streamed to the graphics pipeline through the vertex shader (VS), and decomposed into triangles in the geometry shader (GS). The triangle primitives are sent down the pipeline with scalar values and traversal length as color attributes for the direct volume rendering (DVR) technique, and face normals for the iso-surface rendering (ISO) technique. This process brings a great benefit highly desirable for a hardware-based approach: fitting a volumetric primitive (tetrahedra) to simpler primitives well supported by graphics hardware (triangles). Finally, we use a fragment shader (FS) to evaluate the volume ray integral and to generate the final image.

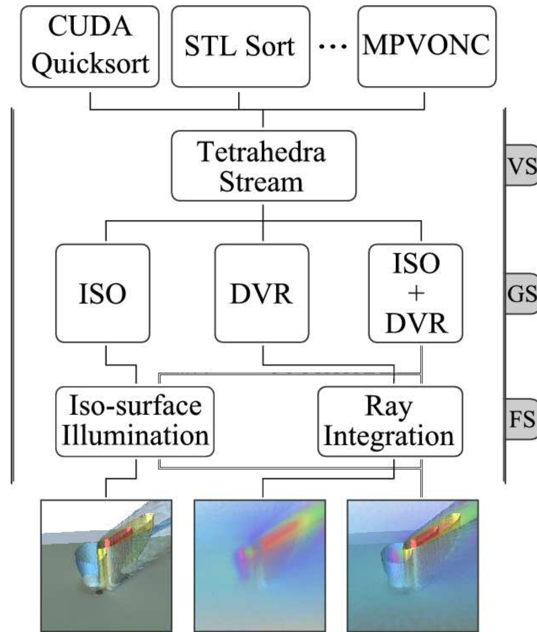


Figure 3.13: HAPT's Framework divided into vertex (VS), geometry (GS) and fragment (FS) shaders. The visualization can be direct (DVR – *direct volume rendering*) or indirect by iso-surfaces (ISO). Any sorting method can be used prior to rendering, for example: quicksort [64] in CUDA; STL-based introsort [65] on the CPU; or the MPVONC algorithm [58] on the CPU.

Algorithm's Pipeline The rendering pipeline is depicted in Figure 3.14. An important point about the data flow is that, since each tetrahedron is processed independently and in a parallel fashion, it fits perfectly within the streaming paradigm. In addition, no auxiliary volume data structure needs to be accessed during rendering as each hardware primitive of HAPT (points, triangles and fragments) contains all information required for processing. This is specially important because it reduces GPU data storage in such a way that there is no restriction to render a volume due to GPU memory requirements. The graphics card memory is limited when compared to the CPU memory, for instance a volume with several million tetrahedra can be rendered by a ray-casting or cell-projection approach on the CPU, but fails to be stored on the GPU when using PTINT [25], IPTINT or HARC [22].

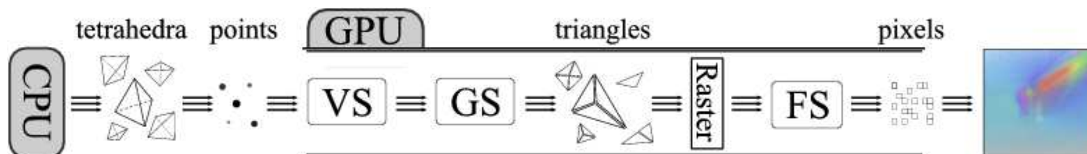


Figure 3.14: HAPT's pipeline: sorted tetrahedra are streamed as point primitives to the GPU; decomposed into triangles in the geometry shader (GS); and finally, during rasterization, the ray integral is computed per fragment to compose the final image.

The streaming feature of HAPT allows it to handle time-varying data trivially as a sequence of static volumes per frame. Moreover, it minimizes the data fetching latency, which usually imposes a high transfer overhead and, consequently, significantly decreases the algorithm's performance. In the remainder of this section each feature and step of the algorithm are presented.

Cells Sorting The HAPT algorithm was tested with four different sorting methods for comparison: the STL-based introsort [65] on the CPU; the MPVONC [58], *Meshed Polyhedra Visibility Ordering for Non-Convex* meshes, on the CPU; the bitonic sort on the GPU using CUDA [66]; and the quicksort on the GPU using CUDA [64]. Except for the MPVONC, the other three strategies perform approximate sort using the tetrahedra centroids.

For the CUDA-based algorithms, the ordered tetrahedron indices are written directly into a GPU *data buffer* or *vertex buffer object* (VBO), avoiding the transfer back to the CPU. In case the data is too large to fit in a VBO, any of the methods can read back from the GPU to the CPU and stream the tetrahedra down the pipeline. Furthermore, the bitonic CUDA sorting method works only with power-of-two arrays, forcing the implementation to enlarge the indices buffer to fit the nearest corresponding size and, therefore, making it slower than the CUDA quicksort counterpart.

For the exact ordering algorithm on the CPU, i.e. MPVONC, it is important to note that it needs two auxiliary data structures: the adjacency list and the precomputed face normals. This extra information can increase the memory consumption on the CPU by around four times the volume size.

There are some degenerate cases where the MPVONC method does not perform a correct visibility ordering. However it works for most meshes [24]. On the other hand, the centroid sorting usually introduces an error, where in some cases the ordering of adjacent tetrahedra can be inverted. Fortunately, the error introduced by this is still very low. As a matter of fact, we noticed no visual difference from the MPVONC, let alone any artifacts.

To support this last statement, we have run a series of tests to estimate the error of centroid sorting in regards to the MPVONC. Table 3.1 presents the average and maximum error for various datasets, described in Chapter 5. The errors are computed per color channel separately, therefore the second column (Maximum Error) is the maximum difference between all channels of all corresponding pixels.

The last column (Different Pixels) gives the percentage of different pixels, that is, pixels between images that are not an exact match in all three RGB channels. For all statistics, only pixels with error above zero are taken into account, thus correct and background pixels are not included. An error of approximately 1.2%, is equivalent to a difference of 3 units in the RGB domain [0, 255] for one specific color channel of a given pixel. Usually the average error is approximately of one single unit, becoming imperceptible for visualization purposes.

Dataset	Maximum Error	Average Error	Different Pixels
blunt	1.961%	0.4069%	6.04%
post	2.353%	0.4245%	33.13%
spx+	1.569%	0.3985%	8.13%
delta	5.098%	0.5895%	14.25%
torso	1.176%	0.3933%	1.51%
fighter	1.569%	0.3943%	2.02%

Table 3.1: Error introduced by the centroid-sorting algorithms when compared against the MPVONC method.

The statistics for Table 3.1 were gathered by rendering with direct volume rendering using both methods (centroid sorting and MPVONC) from at least 100 different and random viewpoints sampled over a sphere. It is also important to notice that the numbers may vary with different transfer functions, even so, we have noticed no discrepancies from the presented values. In this manner, the centroid method is still a good alternative for accelerating rendering speed and cutting down on memory space while introducing a very low error. Since for static data the error is already visually imperceptible even when there are many different pixels, this method is even more adequate for interactive visualization of time-varying data.

Projection To enhance the data throughput to the GPU, a tetrahedron is sent as a vertex with three attributes, that is, the other three vertices are passed as texture coordinates. For each tetrahedron's vertex the attributed scalar value is also passed as the w coordinate. Note that different strategies, such as other geometric primitives, can also be used to send the tetrahedra through the pipeline.

In the geometry shader, the tetrahedron is decomposed into triangles and the three values associated with each vertex, the traversal length l and scalar front s_f and back s_b, are computed using the same scheme as the original PT (see Appendix C for details). To compute the correct tetrahedron projection, we rely on the ternary truth table from PTINT's classification strategy (see Appendix D for details), determining the correct projection case with four cross products and a table lookup.

These three values (s_f, s_b and l) are computed depending on the projected vertex. Remembering, there are two types of projected vertices: the *thick* and *thin* vertices. The thick vertex is defined as the entry point of the tetrahedron where the ray traverses the maximum distance l. Depending on the classification case, the thick vertex may not be one of the four projected tetrahedron's vertices, hence its scalar value has to be interpolated. Analogously, it might be necessary to compute the distance l in cases where l is not one of the edge lengths. Excluding the thick vertex, all others are the projected tetrahedron's vertices, named thin vertices, and have $s_f = s_b$, which are the original scalar value, and $l = 0$, since the ray traverses no distance in these extremities. In the example given in Figure 3.15, v'_0, v'_1 and v'_2 are the thin vertices while v'_3 is the thick vertex. The classification table stores not only the number of generated triangles, but the cases where the traversal distance l has to be computed and scalar values have to be interpolated for the thick vertex.

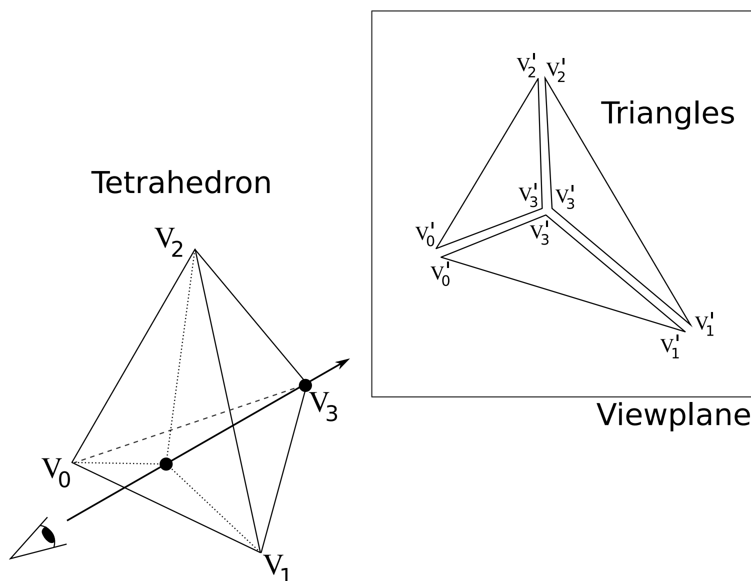


Figure 3.15: Example of class 1 projection of the original PT algorithm, where the projection of the tetrahedron (left) in the viewing plane generates three triangles for rendering.

The scalar values (s_f and s_b) and traversal length (l) are stored as RGB colors for each vertex of all tetrahedra in the output of the geometry shader, either thin or thick vertex. This technique is similar to the one used in PTINT and IPTINT, however in HAPT it serves to pass the information of the triangles generated by the geometry shader to the fragment shader.

Ray Integration When a triangle is rasterized (one tetrahedron cell can be decomposed into one to four triangles), the scalar values and traversal length are interpolated per fragment. This interpolation is an approximation of the precise integration values for each cell used in the integral equation. The interpolated values (s_f, s_b and l) are employed per fragment to compute the ray integration through the partial pre-integration technique, in contrast to using a simple averaging scheme as done by the original PT [6] and GATOR [9]. In this technique the pre-computation of the ray integral equation does not depend on the transfer function, and thus it is pre-compiled within HAPT's application. Both the implementation of the HAPT algorithm as PTINT and its two variants (RPTINT and IPTINT) use the partial pre-integration table pre-compiled into their applications. Inside the fragment shader this table is accessed by two indices computed using the traversal length l and the front and back colors. In turn, these colors can be promptly extracted from the transfer function using the s_f and s_b values. Note that the partial pre-integration technique in fragment shader can be replaced by a better or faster integration method.

Iso-surfaces Rendering The simple and flexible data flow of HAPT permits additional effects that are not trivially implementable with other methods. One example is interactive iso-surface rendering. While most approaches (such as IPTINT) are able to render iso-surfaces on a per-pixel basis (for each fragment determines if the iso-surface is in range), HAPT allows for not only this strategy, but also for an on-the-fly *marching tetrahedra* [67] approach, where for each tetrahedron, one or two triangles are generated corresponding to the iso-surface that crosses that tetrahedron. Using marching tetrahedra, normals can also be extracted to provide illumination effects, unlike IPTINT that uses the gradient of the scalar field as an improvised iso-surface normal. Since the geometry shader processes on a per tetrahedron basis, the iso-surface normals are restricted to faces, resulting in a flat shading. To achieve a smoother Phong-like effect it would be necessary to also carry adjacency information and compute normals per vertex from the incident faces, greatly impacting the memory consumption and rendering performance.

Within the geometry shader the iso-surface can be easily extracted from the tetrahedron and sent for rendering as one or two triangles. This method has two advantages: first, the iso-surface can be interactively defined; and second, it is possible to blend indirect and direct volume rendering generating a hybrid visualization.

Time-Varying Rendering To further demonstrate the algorithm's flexibility, we describe how it can be applied to time-varying datasets, where the volume is a sequence of animation frames, and each frame is a static volume.

As mentioned earlier in this section, the usage of VBOs is optional and not suitable for models that do not fit on the GPU memory; hence, for time-varying datasets, the frames are not stored in memory but streamed as separate static volumes.

Additionally, we apply an early discard test in the vertex shader to remove empty tetrahedra, i.e. cells with all vertices mapped to zero opacity in the transfer function (this discard method is also used in the RPTINT algorithm explained in Section 3.2). Note that the volume must have large empty regions to profit from the additional texture fetches to discard the tetrahedra in vertex shader.

Source Code The HAPT code (with four different sorting methods), using C/C++, C for CUDA and GLSL, is made available with this thesis at:

http://code.google.com/p/hapt.

Chapter 4

Mesh Processing

"The most beautiful experience we can have is the mysterious – the fundamental emotion which stands at the cradle of true art and true science."
– Albert Einstein

In this chapter we introduce the idea of mesh processing augmented by similarity using local exemplars, method that we name *SAMPLE – Similarity Augmented Mesh Processing using Local Exemplars*. The introduced method propagates computation throughout the mesh and it is explained in Section 4.1. The propagation is done employing a similarity space, where similar regions are spatially close, as illustrated in Figure 4.1. We investigate two applications where SAMPLE can be used: detail transfer and mesh parameterization; discussed in Section 4.2. Nonetheless, any mesh processing task, in which replication of processing is suitable, can be augmented using our method.

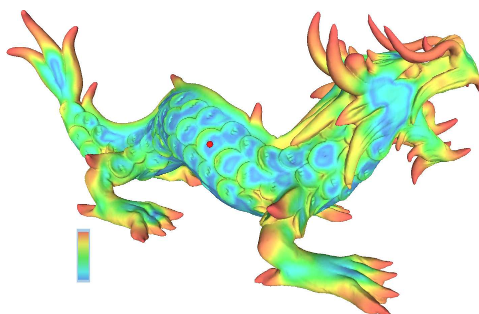


Figure 4.1: Example of similarity space of SAMPLE. The XYZ RGB Asian Dragon is painted by degree of similarity (color scale shown in the bottom-left corner) with the selected vertex (red circle on the dragon scale), from most similar (in blue) to least similar (in red).

4.1 SAMPLE

The *Similarity Augmented Mesh Processing using Local Exemplars* (SAMPLE) method was recently accepted to be published in the *Graphical Models* journal, and constitutes the contribution of this thesis in the mesh processing area.

The main idea behind SAMPLE is to provide a method to propagate computations from an exemplar region to many other similar regions of a mesh. This propagation avoids redundant computation, allowing 3D artists to take advantage of the self-similarity of a model to reduce his/her work. We can cite as an example the automatic painting of similar regions with the same pattern. The concept of similarity in this scenario depends on the mesh properties needed by the mesh processing task. If the processing takes into account the local shape properties, such as normals and curvatures, the similarity descriptor should encode these features. When propagating mesh processing, the descriptor is used to establish non-local neighborhoods where the computation can be replicated throughout the mesh.

The SAMPLE method uses two spaces to accomplish similarity augmented mesh processing: *primal* and *dual* spaces. The primal space defines the regular neighborhood of a local surface patch, given by the mesh connectivity and geometry, and the dual space defines the similarity neighborhood, given by the distance between similarity descriptors. The primal space is widely used by many existing mesh processing tasks, such as Laplacian smoothing by TAUBIN [68]. The dual space, on the other hand, is scarcely used in mesh processing, and it is normally employed as a non-local neighborhood for averaging values [51, 52]. We define and use the dual space in a more comprehensive manner. In our case, the dual neighbors define correspondences across regions in addition to the primal neighbors, aiding in the replication of processing done on an exemplar region to other close regions in the dual space (see an example of dual/similarity space in Figure 4.1).

Once the primal and dual spaces are established, the propagation of mesh processing can be carried out. We illustrate our idea by applying the propagation algorithm in transferring the same details to many similar regions seamlessly, and by reusing the parameterization computed for a patch over the mesh to others similar patches. Both applications are well-suited for propagation through a self-similarity descriptor based on the local shape features, as explained in details in the next section. However, additional mesh processing tasks could use different descriptors, e.g. planar-reflective symmetry [27] or a saliency descriptor [69].

Vertex-based Similarity The dual neighborhood is established by considering the self-similarity of a triangulated mesh, i.e. distances between the shape descriptors of its constituent vertices. These descriptors have to work harmoniously with the mesh processing task at hand. In our scenario, we choose a simple similarity descriptor relying on Euclidean-distance measurements. Each vertex has one such descriptor, which allows for the embedding of the vertex in the dual space. The similarity descriptor is a heightmap computed either by hardware *Z-buffer* or shooting rays from the vertex tangent plane to the mesh surface (see an example in Figure 4.2). The tangent plane is defined by the vertex position and normal. The heightmap grid lying on the tangent plane is aligned with the two principal curvature directions computed at the vertex. The estimation of such differential geometric properties is not always robust. However, this does not significantly impact the efficiency of the descriptors, as the alignment of the grid axes can be described by a simple rotation, and the descriptors are compared in a rotationally-invariant manner.

To compute the heightmap, we consider a bounded square sub-region of the tangent plane, with sides of length $\epsilon = 2.5\%$ of the diagonal of the mesh's bounding box. This sub-region is divided into a 16×16 grid, guiding the casting of rays through each grid cell. We find that these values (grid size and sampling rate) captured the local shape features well in our experiments. Figure 4.2 illustrates one example of a heightmap grid on the spine of the XYZ RGB Asian Dragon model.

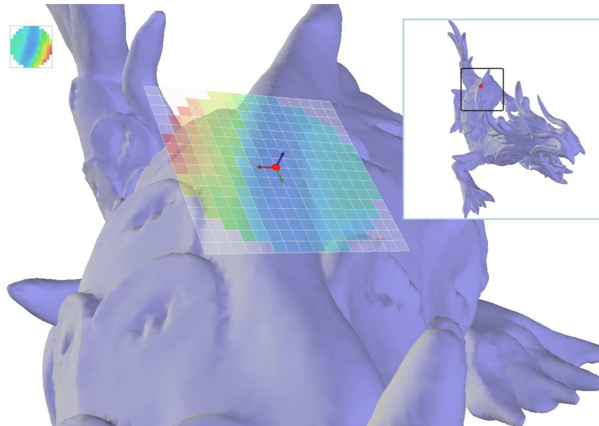


Figure 4.2: Similarity descriptor example for a given vertex (in red). The descriptor is a heightmap (top-left corner) lying on the vertex tangent plane. The red and green arrows are the vertex principal directions, and the blue arrow is the vertex normal. The heights are the Euclidean distances from the plane to the mesh, where the blue and red colors represent low and high values, respectively.

For each grid cell, we cast one ray perpendicular to the tangent plane in both directions, keeping the closest hit as the height value for that cell. Heightmap cells without a hit value and cells outside a fixed radius of ϵ are discarded. The result is a 16×16 resolution image (see one example on the top left of Figure 4.2) describing the surrounding shape of the vertex. This approach is robust to poorly shaped triangles, non-manifoldness and surfaces with holes, since it computes only ray-triangle intersections. A more complex approach using discrete differential geometric properties, such as the HKS [31], could result in a more descriptive signature, but our experiments indicate that such approaches are generally sensitive to triangle quality and topological singularities.

Once we have the heightmap of each vertex, we can measure similarity between two vertices by computing the pixel-wise difference of the two heightmap images. Unfortunately, this leads to poor results. The main problem is that the principal directions of each vertex do not always align the heightmap properly (see Figure 4.3). We address this problem by making use of a basis function which has proved useful in providing rotational invariance in the field of computer vision; the orthogonal Zernike polynomial functions.

The straightforward approach for the alignment problem is to compute the difference for every rotation and choose the minimum value as the similarity. While this brute-force approach would yield the correct result, it is very inefficient. Instead of this approach, we convert each heightmap to Zernike coefficients, named after the work of ZERNIKE [70], and compare these coefficients rather than the image pixels.

Intuitively, the action of converting an image to Zernike coefficients (or *moments*) means finding the values that when multiplied by each “base image” approximate the original image. For example, in Figure 4.3 the original image A, which corresponds to the heightmap of the vertex A on the spine of the XYZ RGB Asian Dragon model, is decomposed in a sequence of “base images”, the so-called Zernike polynomials, that are multiplied by a set of coefficients. This set is the result of the action of converting a heightmap to Zernike coefficients. The advantage of using Zernike coefficients, instead of the heightmap image of a vertex, is that they are rotationally invariant by the image that originate them.

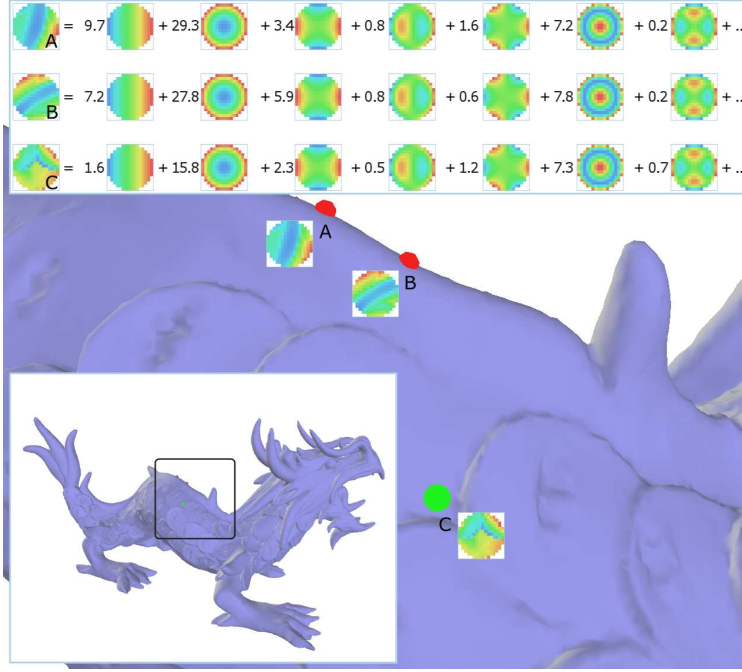


Figure 4.3: Zernike expansions for heightmaps. Only the magnitudes of the complex values for the coefficients (top three row numbers) and Zernike polynomials (images after the equal signs) are shown. Although vertices A and B (in red) are similar, their heightmap images have improper alignment based on principal directions. By converting them to Zernike coefficients, the heightmaps are correctly compared. Vertex C (in green) is also correctly classified as different than vertices A and B.

Specifically, the Zernike polynomials constitute an orthogonal basis for functions defined on the unit disk. Each Zernike polynomial, V_p^q, has an associated order p and repetition q, and they are defined over the domain $D = \{(p,q) \mid q \in \mathbb{Z}, p \in \mathbb{Z}^{\geq 0}, |q| \leq p, |p-q| \in \mathbb{Z}^{\text{even}}\}$ as follows:

$$V_p^q(\rho,\theta) = R_p^q(\rho)e^{iq\theta} \tag{4.1}$$

Where R_p^q is the radial polynomial given by:

$$R_p^q(\rho) = \sum_{\substack{k=|q| \\ |p-q| \text{even}}}^{p} \frac{(-1)^{\frac{p-k}{2}} \frac{p+k}{2}!}{\frac{p-k}{2}!\frac{k-q}{2}!\frac{k+q}{2}!}\rho^k \tag{4.2}$$

To obtain the Zernike coefficients of a function $f(x,y)$, we apply:

$$z_p^q(f) = \frac{p+1}{\pi} \iint_{x^2+y^2\leq 1} (\bar{V_p}^q)(x,y)f(x,y)dxdy \tag{4.3}$$

Where $(\bar{V_p}^q)$ denotes the complex conjugate of (V_p^q). The Zernike polynomials form a basis upon which an image f can be projected. The result of this projection is the Zernike moments of the image, the magnitudes of which are invariant to rotation. The reader can refer to the paper of KHOTANZAD and HONG [71] for a detailed explanation of Zernike moments applied to the pattern recognition context in images.

For our heightmap images (with 16×16 resolution), we project $f(x, y)$ onto 25 Zernike polynomials (corresponding to non-negative repetitions and up to the $8th$ polynomial order) resulting in a vector $\mathbf{z}_i$ of Zernike moments for each vertex v_i. We have noticed that 25 coefficients are sufficient to represent the vertex's descriptor. Figure 4.3 shows an example of two heightmap images (A and B) with an improper alignment using the principal directions, whereas the Zernike coefficients of the two images have close values, correctly classifying as similar the two vertices on the spine of the Asian Dragon model. The processing time spent in computing the brute-force approach is three orders of magnitude greater than the computation using Zernike coefficients. More specifically, employing Zernike coefficients reduces the similarity measurement of entire meshes time from hours to seconds.

Region-based Similarity With the two techniques explained in the previous section, we have a better similarity measurement among vertices. The problem that remains is that the dual space resulting from measuring similarities between isolated vertices may have misplaced correspondences for nearby vertices, and it may ignore distinctive shape features in favor of large flat regions (see example in Figure 4.4, left). We solve this problem by considering entire neighborhoods rather than single vertices when building the dual space. For each vertex, we apply Gaussian weights to the Zernike coefficients, adding the coefficients of nearby vertices within a fixed radius of ϵ. The Gaussian filter cut-off is set to be at a distance of 2σ, where σ is set to $\epsilon/2$ to include all vertices inside the region considered by the heightmap.

Recall that we have defined $\mathbf{z}_i$ to be the vector of Zernike coefficients for the vertex v_i. Our region-based comparison method uses the primal neighborhood of v_i, denoted by $\mathcal{N}(v_i, \sigma)$, which includes vertices within a distance σ of v_i. To apply our Gaussian-weighted comparison scheme, we consider a new vector of Zernike coefficients for each vertex defined as follows:

$$\mathbf{z}_i^\sigma = \frac{\sum_{v_j \in \mathcal{N}(v_i, 2\sigma)} \mathbf{z}_j e^{-\frac{\left|v_i - v_j\right|^2}{2\sigma^2}}}{\sum_{v_j \in \mathcal{N}(v_i, 2\sigma)} e^{-\frac{\left|v_i - v_j\right|^2}{2\sigma^2}}} \tag{4.4}$$

The $\mathbf{z}_i^\sigma$ is defined as a Gaussian-weighted average of the Zernike coefficient vectors of all vertices within a radius 2σ of v_i. These newly formed vectors of Gaussian-

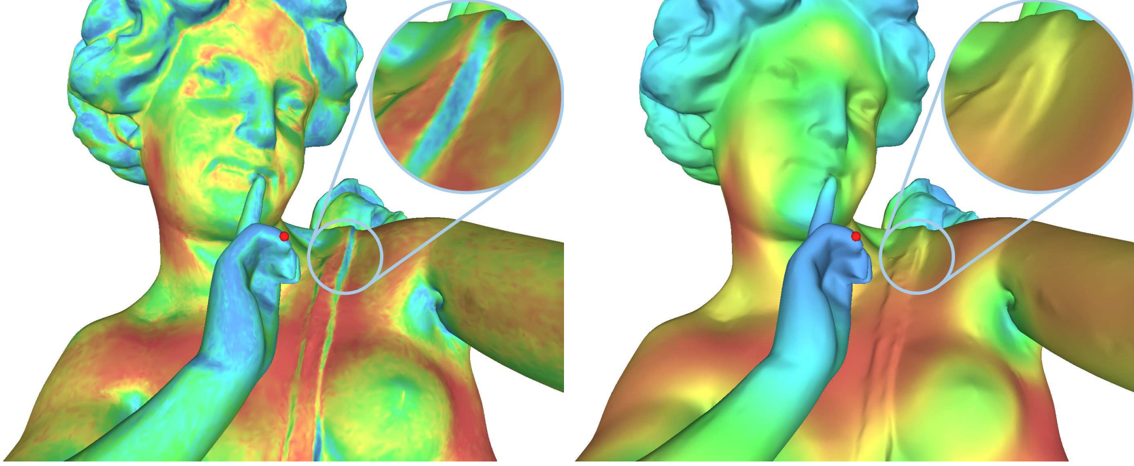


Figure 4.4: Difference between vertex-to-vertex (left) and region-to-region (right) comparison methods. The similarity measurement based on individual vertices yields the incorrect comparison of the selected vertex (in red) to the strap region close to the shoulder (top-right corner). The model is painted from most similar (in blue) to least similar (in red).

weighted Zernike coefficients replace the original vectors, resulting in the generation of a smooth and high-quality dual space. Figure 4.4 and 4.5 illustrate the *3DScanCo Angel* and *Dama* models using vertex-based comparison and the Gaussian-weighted region method.

It is important to note that this region-based comparison benefits greatly from the rotational invariance of our descriptor. While this property is not strictly necessary to define a descriptor, it allows for the agglomeration of local descriptors without concerning with their alignment. Essentially, the rotational invariance of the Zernike coefficients gives us the ability to build our region-based descriptors simply by taking a Gaussian-weighted average of the Zernike coefficients in the region surrounding a vertex.

In the absence of the rotational invariance property, an attempt to build a region-based descriptor from an agglomeration of vertex-based descriptors (e.g. geodesic fans) must address the challenging question of how these descriptors should be aligned and combined. A brute-force approach is difficult, because attempting to find the best alignment between all vertex descriptors in the region is a significantly more difficult problem than simply finding the minimum distance alignment between two descriptors. Furthermore, the minimum distance alignment between the neighboring vertex descriptors may not even constitute the proper alignment of the descriptors. In this case, an explicit attempt to model the neighborhood structure needs to be made; thus a comparison method is needed on these neighborhood descriptors which may be significantly more complicated than their vertex-based counterparts.

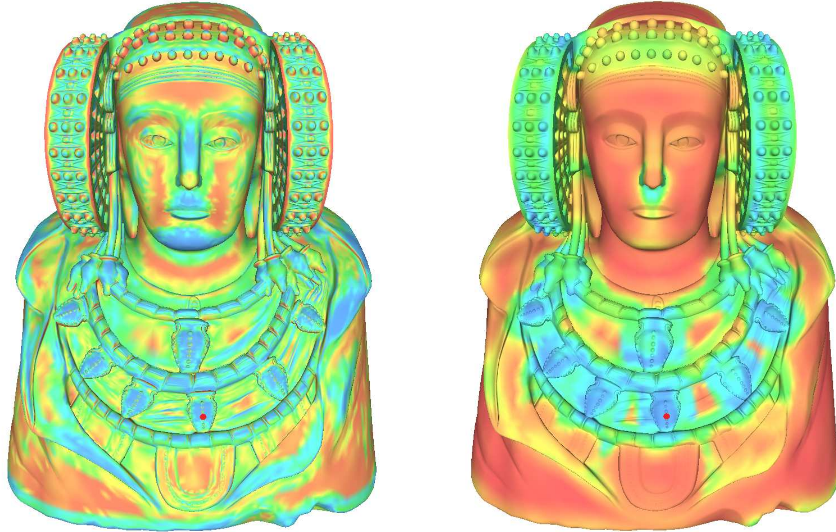


Figure 4.5: The dual space of the Dama model further illustrates the difference from vertex-based (left) to region-based (right) comparisons. Excessive blue color in the left image shows that the vertex-based technique is not as discriminative as the region-based technique shown in the right image.

Establishing the Dual Space After applying the techniques explained previously, we obtain 25 coefficients for each vertex, which constitute its embedding coordinates in the dual space. This space is our similarity space and can simply be viewed as $\mathbb{R}^{25}$, where similar vertices can be found by searching for nearest neighbors in this space, under an Euclidean metric, for any given vertex.

The similarity distance s in the dual space between vertices v_i and v_j is defined as follows:

$$s(v_i, v_j) = d(\mathbf{z}_i^\sigma, \mathbf{z}_j^\sigma) \tag{4.5}$$

Where $d(\cdot, \cdot)$ denotes the Euclidean-distance metric. The mesh dual space depends only on the Gaussian-weighted Zernike coefficients of each vertex, $\mathbf{z}_i^\sigma$, which can be pre-computed and stored as an auxiliary data structure for similarity measurements. The dual neighbors of a vertex v_i is defined as the set of vertices under a threshold τ where $s(v_i, v_j) < \tau$. Figure 4.6 shows a number of dual space neighbors of one vertex on the scale of the Asian Dragon model, the vertices in the primal neighborhood are illustrated for comparison.

Propagating Mesh Processing The SAMPLE algorithm propagates mesh processing by using the dual space and a *local exemplar*. An exemplar region is any region of the mesh with desired features, which is used to guide the propagation of a target processing task. For example, to paint all the scales of the XYZ RGB Asian Dragon model with the same drawing, one of the scales can be selected as the

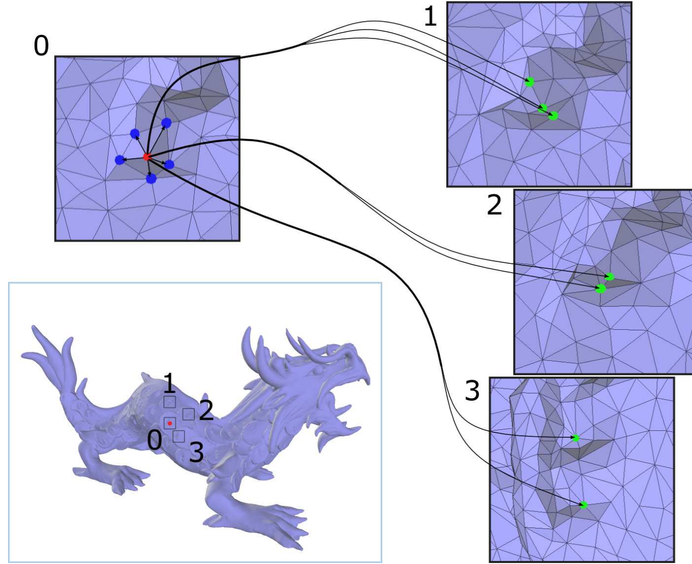


Figure 4.6: Illustration of the primal and dual spaces. Several primal neighbors of the selected vertex (in red) on a dragon scale are shown in blue (box 0). While some of the dual neighbors are shown in green (boxes 1, 2 and 3).

exemplar region and, as the texture painting occurs, all the regions locally similar to the exemplar are painted with the same pattern (see Figure 4.7). The disadvantage of this approach is that it relies upon the size of the local exemplar region, which is directly related to the size of the heightmap descriptor explained in the previous section. If the local exemplar region is much smaller or larger than the heightmap, then the dual space fails to capture the distinctive features of the local shape, providing the incorrect similar neighbors upon which the processing is propagated. In these cases, the dual space has to be recomputed considering an ϵ value chosen to match the size of the desired region and yield the correct result.

The dual space plays the main role in the propagation of mesh processing. It is responsible for describing the similarity correspondences among vertices, which are used to propagate computation across similar regions. After a local exemplar is chosen, we can determine its dual neighbors. These neighbors are those vertices whose dual space embedding reside within a user-defined distance threshold τ from the embedding of the exemplar's center vertex. Then, the operations performed on the exemplar can be carried out on these other similar regions. The similarity threshold defines which regions are affected by the propagation. Smaller threshold values affect fewer regions, while larger values affect more regions.

Although the comparison method used to build our dual space considers entire mesh regions, the dual neighborhood is defined among mesh vertices. The dual neighbors of a given vertex may also be close to this vertex in the primal space. This scenario can compromise the propagation of mesh processing. We solve this problem by considering only neighboring vertices in the dual space whose primal

regions do not overlap each other. That is, the size of the exemplar region defines the minimum distance in the primal space that we will require between any pair of vertices before we consider them for the propagation of processing.

After these considerations, we can now describe how the propagation of mesh processing between the local exemplar and a target region is actually performed. We define a local coordinate frame for each vertex undergoing the propagation in order to replicate processing consistently. This frame is based on the same vectors used to compute and align the heightmap descriptor over the vertex tangent plane, and it is used to assign local coordinates (u, v, w) for each vertex inside the affected regions. These coordinates replace the original coordinates (x, y, z) while processing similar regions, creating a common coordinate system throughout the propagation. The size of each affected region is defined by the exemplar region, where the desired local shape resides. Since our similarity measurement is rotationally invariant and the heightmaps of the affected regions may not be aligned, we rotate the (u, v) coordinates by an angle, α, which is chosen to achieve the minimum similarity difference between the affected and exemplar regions. Although a technique exists to retrieve the rotational angle using Zernike, technique of REVAUD *et al.* [72], in our experiments, this method yields several local minima for α, degrading the propagation. Since we generally have a small number of regions among which to propagate results, we compute the α angle using the brute-force approach discussed above, i.e. computing the heightmap differences for all possible rotations and choosing the angle resulting in the minimum value.

Finally, the local coordinates are used to find the nearest vertices in the exemplar region and interpolate an approximate result value of any processing done on the local exemplar for the vertex on the target affected region. The quality of this approximation depends on the processing performed and the quality of the mesh samples. A more regularly sampled mesh generally yields better results when propagating processing operations than a poorly sampled one. In the next section, we illustrate this idea using our SAMPLE algorithm in two applications: detail transfer and mesh parameterization.

4.2 Applications

In this section two application of the SAMPLE method will be presented to illustrate the usage of the dual space and the propagation ideas. The first (see Figure 4.7) replicates the parameterization computed for one exemplar region to several other similar regions, preserving scale and local orientation of the parameterization domain. The second (see Figure 4.8) propagates a given detail mask to many similar regions, preserving local consistency.

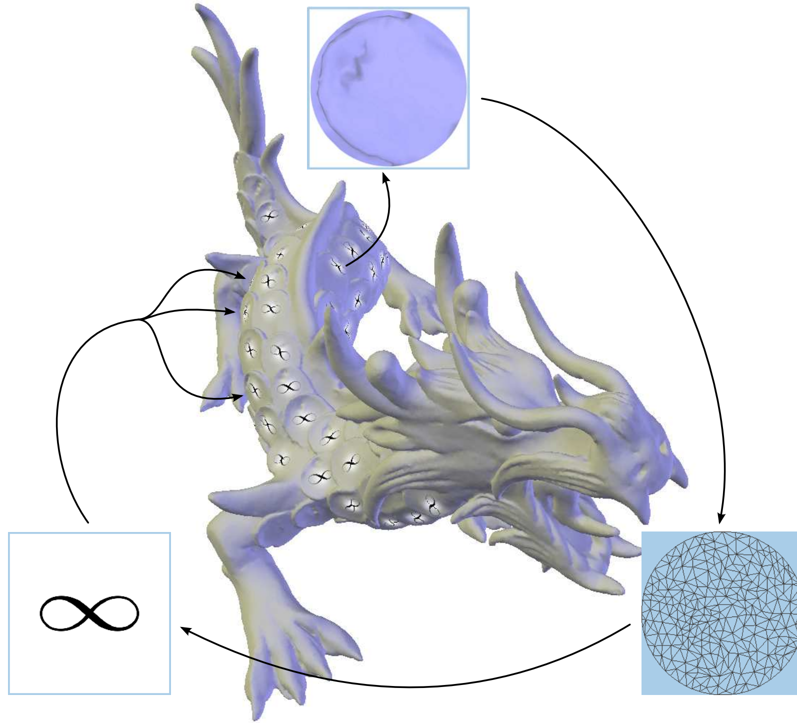


Figure 4.7: SAMPLE application: mesh parameterization. One of the dragon scales is elected to be the exemplar region (top); the exemplar is parameterized using a standard procedure (bottom right); the infinity logo image (bottom left) is used to texture the similar dragon scales seamlessly.

The first application aims to reuse a local mesh parameterization taking advantage of the dual space to replicate the computed parameters. We perform a simple parameterization, using the *Floater Mean Value Coordinates* algorithm of FLOATER [73], on the selected exemplar region, and subsequently visit a number of similar regions which reside within a given similarity distance threshold in the dual space. For each similar region, we assign values from the exemplar parameterization corresponding to the local coordinates (u, v, w). In this way, the same parameterization domain is shared among regions with similar local shape. Figure 4.7 shows the replication process applied to the XYZ RGB Asian Dragon model.

Note in Figure 4.7 that a small number of scales are not affected by the process, since they consist of vertices whose dual space embedding reside beyond the specified distance threshold from the exemplar scale. We chose this distance threshold because it allows us to select most of the similar scales without producing false positives, i.e. similar non-scale regions, which tend to occur at greater similarity distance thresholds. Also note that, although the replication procedure induces a maximum error displacement of 0.047 units on the parameterization disk domain, the textures are not visibly distorted.

The second application aims to use the dual space to propagate mesh editing throughout meaningful regions. We use a simple editing mask, a Gaussian-weighted

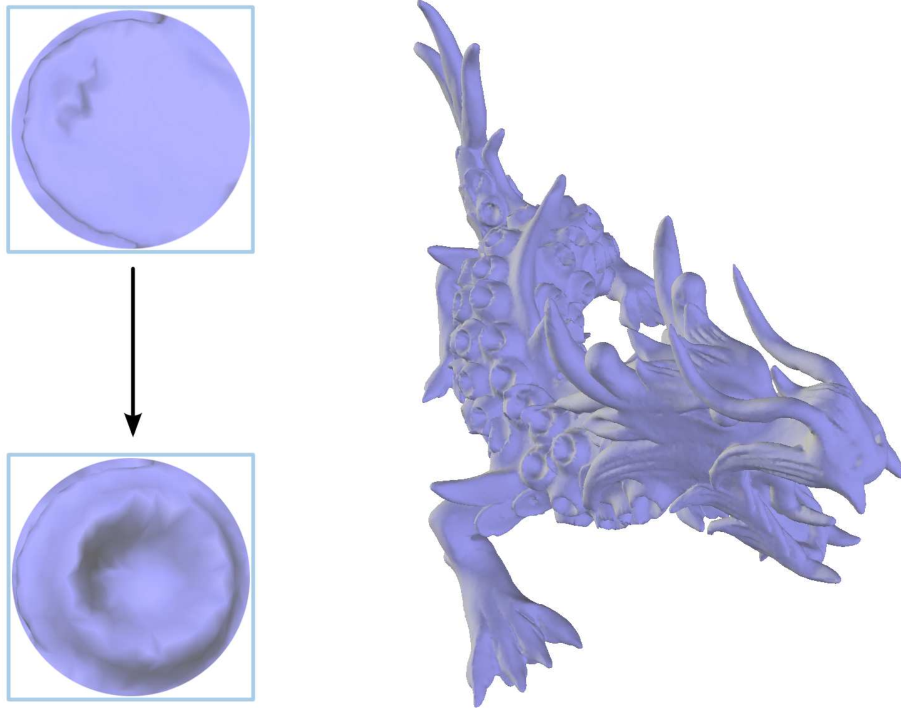


Figure 4.8: SAMPLE application: detail transfer. One exemplar region (top-left corner) of a dragon scale is edited to resemble a crater (bottom-left corner). Our similarity augmented algorithm propagates the editing result naturally over the similar scales.

extrusion of a fixed radius (shown in the bottom-left corner of Figure 4.8), applied to the same exemplar dragon scale used in the first application. We also use the same similarity distance threshold to affect the identical scales, illustrating that our SAMPLE algorithm can be employed in a similar manner for a broad range of mesh processing tasks. In this application, the mesh that results from the propagation of the editing performed on the single exemplar region is identical to the mesh which would result if all of the similar regions had been edited manually. This illustrates how the SAMPLE system can be used to reduce the burden on an artist, who would otherwise have to manually edit each of the similar regions to obtain the same result.

Chapter 5

Results

"It is through science that we prove,
but through intuition that we discover."
– Jules Henri Poincaré

In this chapter we shall present results for the algorithms of this thesis. In the two previous chapters were described four improved algorithms for volume rendering and introduced one mesh processing method. The five techniques presented were:

- VF-Ray-GPU – *Visible-Face Driven Ray Casting implemented on the GPU*;
- RPTINT – *Regular Projected Tetrahedra with Partial Pre-Integration*;
- IPTINT – *Improved Projected Tetrahedra with Partial Pre-Integration*;
- HAPT – *Hardware-Assisted Projected Tetrahedra*;
- SAMPLE – *Similarity Augmented Mesh Processing using Local Exemplars*.

The implementation of all techniques was performed using the following languages: C/C++, C for CUDA and GLSL; and employing the following libraries: OpenGL [1], GLUT [74], CGAL [75], *Boost* [76], VCGLib [77] and LCGtk [78]. Performance measurements were made using different computers and graphics cards, explained in each of the techniques.

Section 5.1 presents results for the volume rendering algorithms, and Section 5.2 presents results for the mesh processing method.

5.1 Volume Rendering

The experiments performed for the volume rendering algorithms presented in this thesis used the following irregular volume data: *Blunt Fin* (*blunt*); *Combustion Chamber* (*comb*); *Liquid Oxygen Post* (*post*); *Super Phoenix* incremented (*spx+*); *Delta Wing* (*delta*); *Human Torso* (*torso*); *F-16 Jet Simulation* (*f16*); *Langley Fighter* normal (*fighter*) and incremented (*fighter+*). In addition to these datasets, the following regular volume data were also used: *Fuel Injection* (*fuel*); *Electron Distribution Probability* (*neghip*); *Tooth CAT scan* (*tooth*); *Foot X-Ray* (*foot*); *Skull CAT scan* (*skull*); and *Aneurysm X-Ray* (*aneurysm*). Finally we used a time-varying volume data called: *Turbulent Jet* (*turbjet*). The datasets used in this thesis were acquired from different sources, however most of them are available at:

http://www.volvis.org.

The validation of the presented algorithms were performed by comparing them with the following state-of-art algorithms:

- VF-Ray – Original *Visible-Face Driven Ray Casting* algorithm [53];
- PTINT – Original *Projected Tetrahedra with Partial Pre-Integration* algorithm [25];
- GATOR – *GPU-Accelerated Tetrahedra Renderer* [9];
- VICP (GPU) (CPU) – *View-Independent Cell Projection* (implemented on the GPU and on the CPU) [17];
- VICP (Balanced) – *VICP* with GPU-CPU balancing [12];
- HARC – *Hardware-Based Ray Casting* [22];
- HARC (INT) – *HARC* with partial pre-integration [23];
- HAVIS – *Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection* [8];
- HAVS – *Hardware-Assisted Visibility Sorting* [18].

The algorithms used for comparison were implemented or acquired from their authors (see Chapter 2 for a general explanation of these algorithms). This thesis uses one or more algorithms listed depending on the technique to be compared. At the end of this section, a direct comparison among the techniques presented in Chapter 3 is outlined.

VF-Ray-GPU The experiments of the VF-Ray-GPU algorithm were conducted on an Intel Pentium Core 2 Duo 6400 with 2.13 GHz per processor and 2 GB RAM. The graphics card used was a nVidia GeForce 8800 Ultra with 768 MB of memory.

The evaluation of our algorithm was done against recent hardware-based algorithms. VF-Ray-GPU was also compared with the original version of VR-Ray that runs solely on the CPU. The idea behind this last comparison is to: first, examine how much the usage of the graphics hardware can improve the performance over a CPU implementation; and second, analyze the gain we obtained by using different and more efficient data structures to the GPU presented in this thesis in Section 3.1.

The datasets used in the experiments were: "blunt", "post", "fighter+" and the "f16". Table 5.1 shows several properties of the volumetric data tested: the number of vertices (# *Verts*), faces, external faces (# *Boundary*) and tetrahedra (# *Cells*); where K and M mean thousands and millions, respectively. As can be seen in this table, two small datasets, "blunt" and "post", and two large datasets, "fighter+" and "f16", were used. The final image resolution generated for small datasets was 512×512 and for large datasets 2048×2048.

Dataset	# Verts	# Faces	# Boundary	# Cells
blunt	41 K	381 K	13 K	187 K
post	109 K	1 M	27 K	513 K
f16	1.1 M	12.9 M	309 K	6.3 M
fighter+	1.9 M	22.1 M	334 K	11.2 M

Table 5.1: Properties of the tested datasets.

The following algorithms were used for comparison: VICP of WEILER *et al.* [17] is a hybrid algorithm that projects each volume cell, performing integration inside each cell using ray casting; HARC of WEILER *et al.* [22] is based on BUNYK *et al.* [11]'s algorithm with the addition to store the face and adjacency information on the GPU and to compute the contribution of each cell by accessing a 3D texture with pre-integration values; HARC (INT) of ESPINHA and CELES [23] is an extension of the HARC algorithm with partial pre-integration, which employs alternative data structures to reduce the memory consumption; VF-Ray of RIBEIRO *et al.* [53] is the original version of the algorithm presented in this thesis on the CPU, explained in Appendix B.

In Table 5.2, the VF-Ray-GPU algorithm is compared with other graphics card algorithms, using the following memory aspects: number of bytes per tetrahedron (*Bytes/Tet*); number of bytes per pixel (*Bytes/Pixel*); and the total number of megabytes spent in normal or partial pre-integration techniques (*Pre-Int*). These numbers indicate the memory requirement of each algorithm, given the size of the rendered volume and the precision of the generated image.

Algorithm	*Bytes/Tet*	*Bytes/Pixel*	*Pre-Int*
VICP	456	–	16
HARC	160	96	16
HARC (INT)	96	96	1
VF-Ray-GPU	38	–	–

Table 5.2: Memory aspects of the VF-Ray-GPU algorithm and other recent algorithms in hardware.

Table 5.3 shows the time and memory usage results for the following algorithms: VICP, HARC, HARC (INT) and VF-Ray-GPU; for the two small datasets. The compared algorithms on the GPU fail to visualize the large datasets, due to memory limitation, preventing the comparison of them with VF-Ray-GPU for these data. Two columns for each dataset summarizes the total memory footprint (in kilobytes) and time spent (in milliseconds) to render a frame (in a viewport of 512×512 pixels).

Algorithm	blunt		post	
	Memory (KB)	*Time (ms)*	*Memory (KB)*	*Time (ms)*
VICP	118,524	190	249,928	546
HARC	72,267	18	123,245	33
HARC (INT)	22,636	32	50,248	51
VF-Ray-GPU	7,029	186	19,494	370

Table 5.3: Comparison of the VF-Ray-GPU algorithm and other recent GPU algorithms.

In Table 5.4, the VF-Ray-GPU algorithm and the original VF-Ray are compared. The table shows the memory footprint (in kilobytes) and the total execution time (in millisecond) for the two large datasets. As we can observe in this table, in comparison with the original VF-Ray, our algorithm runs about 7 times faster and uses about 50% less memory. The performance gain is due to the parallel nature of our ray-casting algorithm, while the data restructuring presented improves the memory footprint spent in graphics card.

Dataset	Memory (MB)		Time (ms)	
	CPU	GPU	CPU	GPU
fighter+	876	426	58326	10035
f16	499	239	51235	8815

Table 5.4: Comparison between the original VF-Ray algorithm on the CPU and the VF-Ray-GPU algorithm presented in this thesis.

Figure 5.1 presents rendering results of our algorithm. The four datasets used for testing were rendered.

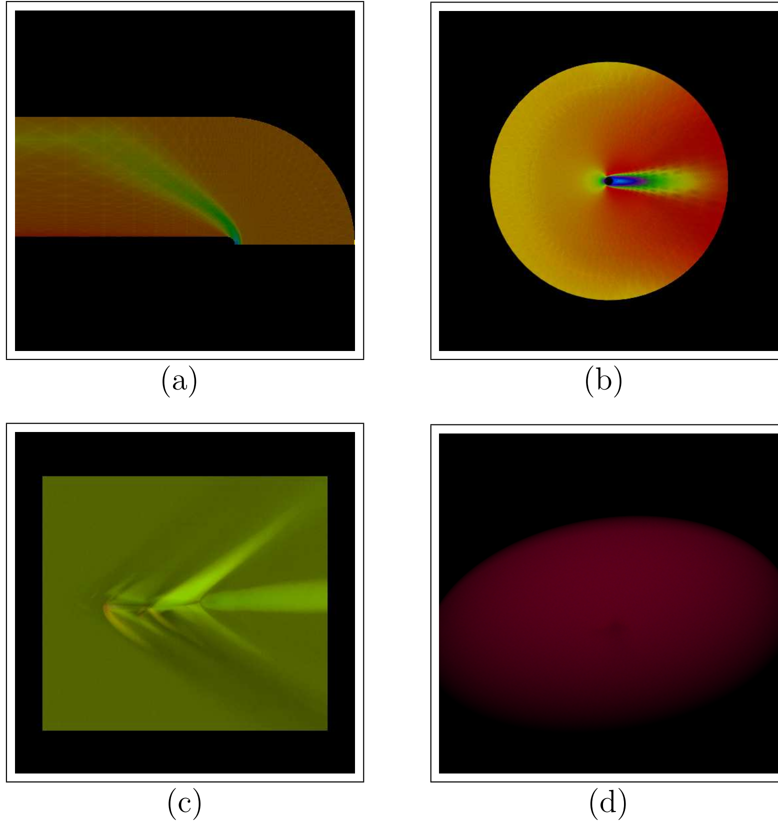


Figure 5.1: Images generated by the VF-Ray-GPU algorithm using the following tested datasets: (a) “blunt”, (b) “post”, (c) “fighter+” and (d) “f16”.

RPTINT The performance of the RPTINT algorithm was measured on an Intel Pentium IV 3.6 GHz, 2 GB RAM, with a nVidia GeForce 6800 256 MB graphics card connected to a PCI Express 16x bus.

The results of the regular volume data used for testing are shown in Table 5.5. The performance measures were obtained using a viewport of 512×512 pixels with the volume in constant rotation. This rotation procedure is important to change between different projection classes, since parallel rendering generates less triangles.

Dataset	# Verts	# Tet	*fps*	*M Tet/s*
fuel	262 K	1.2 M	70.78	88.5 (5.99)
tooth−	1 M	5 M	1.22	13.1 (6.30)
tooth	10 M	52 M	0.24	12.7 (6.61)
foot	16 M	83 M	0.81	67.7 (6.23)
skull	16 M	83 M	0.61	51.7 (6.25)
aneurysm	16 M	83 M	2.42	201 (5.35)

Table 5.5: Performance measurement of the RPTINT algorithm for different regular volume datasets. The “tooth−” result corresponds to the original “tooth” tomography clipped with our tool (discussed in Section 3.2).

We use five different datasets to measure RPTINT. One simulation of a fuel injection (“fuel”), one computed tomography of a tooth (“tooth”) and a skull (“skull”), and the x-ray scan of a human foot (“foot”) and an aneurysm (“aneurysm”). Four models rendered with our algorithm were shown in Figure 5.2 (while the “skull” model was shown in Figure 1.2).

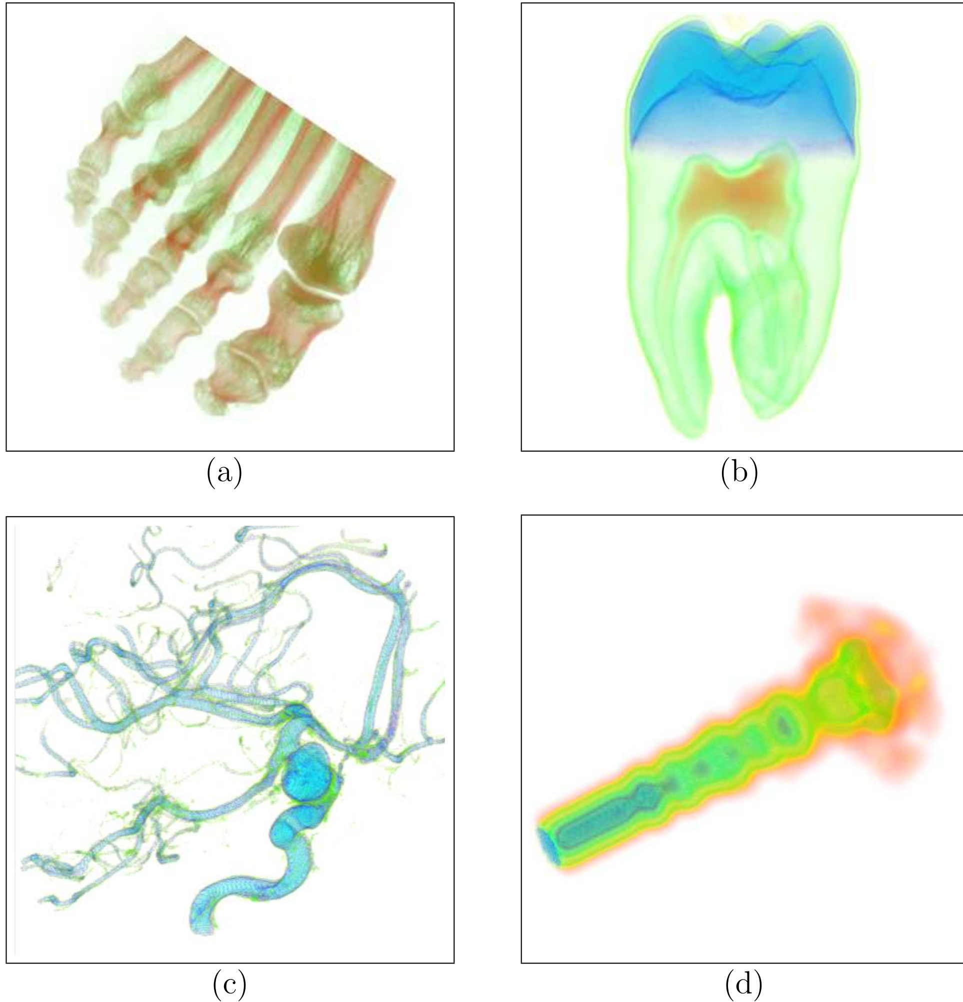


Figure 5.2: Images generated by the RPTINT algorithm using the following regular volume data: (a) “foot”, (b) “tooth”, (c) “aneurysm” and (d) “fuel”.

In Table 5.5, the number of vertices (# Verts) and tetrahedra (# Tet) depends on the dimension of the original regular dataset. Performance measures are given in frames per second (*fps*) and millions of tetrahedra per second (*M tet/s*) for each dataset. In fact, this last column contains two values: the first is the nominal number of tetrahedra per second, including those which do not contribute to the final image generation, while the second is the effective number, i.e. the tetrahedra actually rendered. This effective number (given in parentheses) counts only the tetrahedra responsible to generate the final image, since the volunits with zero opacity are discarded.

The timings for the vertex array allocation or setup, used to send the volume information to the GPU, and rendering are given in Table 5.6 (corresponding to the third and fourth steps of the RPTINT algorithm). In our algorithm, the average time spent in rendering is more than 60% of the total time, demonstrating that RPTINT generates images from volumetric data with a performance close to the limit of the graphics card.

Dataset	Setup	Render	% Total
fuel	0.005 s	0.009 s	64.28 %
tooth	1.377 s	6.484 s	82.47 %
foot	4.556 s	9.074 s	66.57 %
skull	4.606 s	8.593 s	65.10 %
aneurysm	0.199 s	0.210 s	51.34 %

Table 5.6: Time spent (in second) on the two final steps of RPTINT: vertex, color, normal and count arrays setup; and volume rendering. The time percentage spent in rendering when compared with the total time is given in the last column.

IPTINT Performance measures of IPTINT were made on the same PC used in RPTINT, an Intel Pentium IV 3.6 GHz, 2 GB RAM, with a nVidia GeForce 6800 256 MB graphics card connected to a PCI Express 16x bus.

The volumetric datasets used to test IPTINT were: “blunt”, “comb”, “post”, “spx+”, “fuel” and “neghip”. Images generated by our algorithm of these datasets can be seen in Figures 5.3, 5.4, and 5.5.

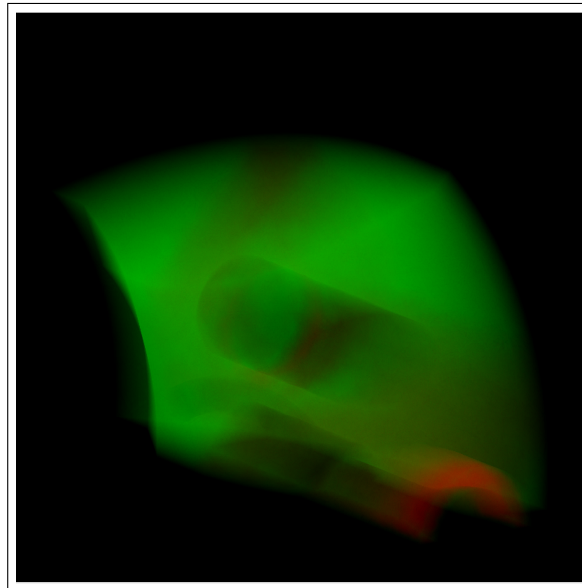
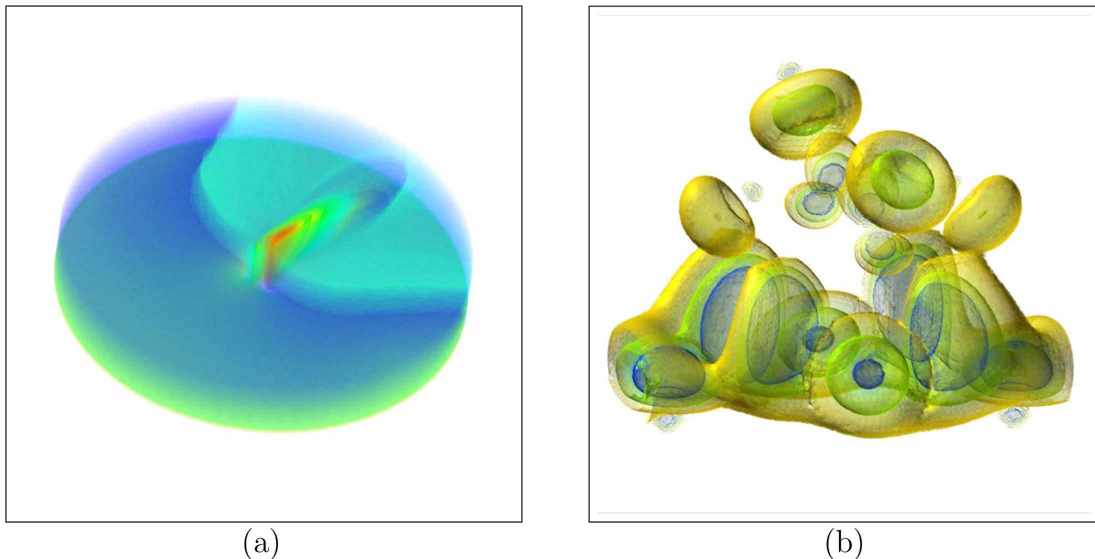


Figure 5.3: Image from the “spx+” dataset generated by IPTINT.

The timings are given using a 512×512 pixel viewport and considering that the model is constantly rotating. Table 5.7 compares our algorithm (IPTINT) with other state-of-art volume rendering algorithms on the graphics hardware.

Algorithm	blunt	post
PTINT	10.76 *fps*	4.35 *fps*
IPTINT	4.97 *fps*	2.09 *fps*
GATOR	4.07 *fps*	1.51 *fps*
VICP (GPU)	5.20 *fps*	1.93 *fps*
VICP (CPU)	1.82 *fps*	0.57 *fps*
VICP (Balanced)	4.10 *fps*	1.11 *fps*
HARC	4.47 *fps*	8.63 *fps*
HARC (INT)	4.94 *fps*	5.93 *fps*
HAVIS	2.36 *fps*	0.79 *fps*
HAVS (k=2)	6.09 *fps*	3.09 *fps*
HAVS (k=6)	3.45 *fps*	2.09 *fps*

Table 5.7: Comparison of the IPTINT algorithm and other approaches. The base PTINT algorithm [25] is identical to run IPTINT without support to render iso-surfaces, that is, only using the partial pre-integration technique.



(a) (b)

Figure 5.4: Images generated by IPTINT using the following datasets: (a) "post" and (b) "neghip".

Table 5.8 further specifies the number of vertices (# Verts) and tetrahedra (# Tet) for each tested dataset, and the average number of frames per second (*fps*) and tetrahedra per second (*Tet/s*) of our algorithm (IPTINT) in three variants: with the original PT method of averaging scalar values (*Basic*); partial pre-integration technique (*+INT*); and using both partial pre-integration and iso-surfaces rendering (*+ISO*).

Dataset	Size		*Basic*		*+INT*		*+ISO*	
	# Verts	# Tet	*fps*	*M Tet/s*	*fps*	*M Tet/s*	*fps*	*M Tet/s*
blunt	40 K	187 K	12.75	2.38	10.76	2.01	4.97	0.93
comb	47 K	215 K	11.12	2.38	8.98	1.92	3.71	0.79
post	110 K	513 K	4.91	2.51	4.35	2.23	2.09	1.07
spx+	150 K	828 K	4.68	2.55	4.52	2.47	1.23	1.02
fuel	262 K	1.25 M	26.01	2.10	22.99	1.86	9.95	0.80
neghip	262 K	1.25 M	3.59	2.34	3.11	2.03	1.27	0.83

Table 5.8: Dataset sizes and average time of the IPTINT algorithm for three different renderings: basic average scalar method; partial pre-integration technique; and partial pre-integration and iso-surface extraction.

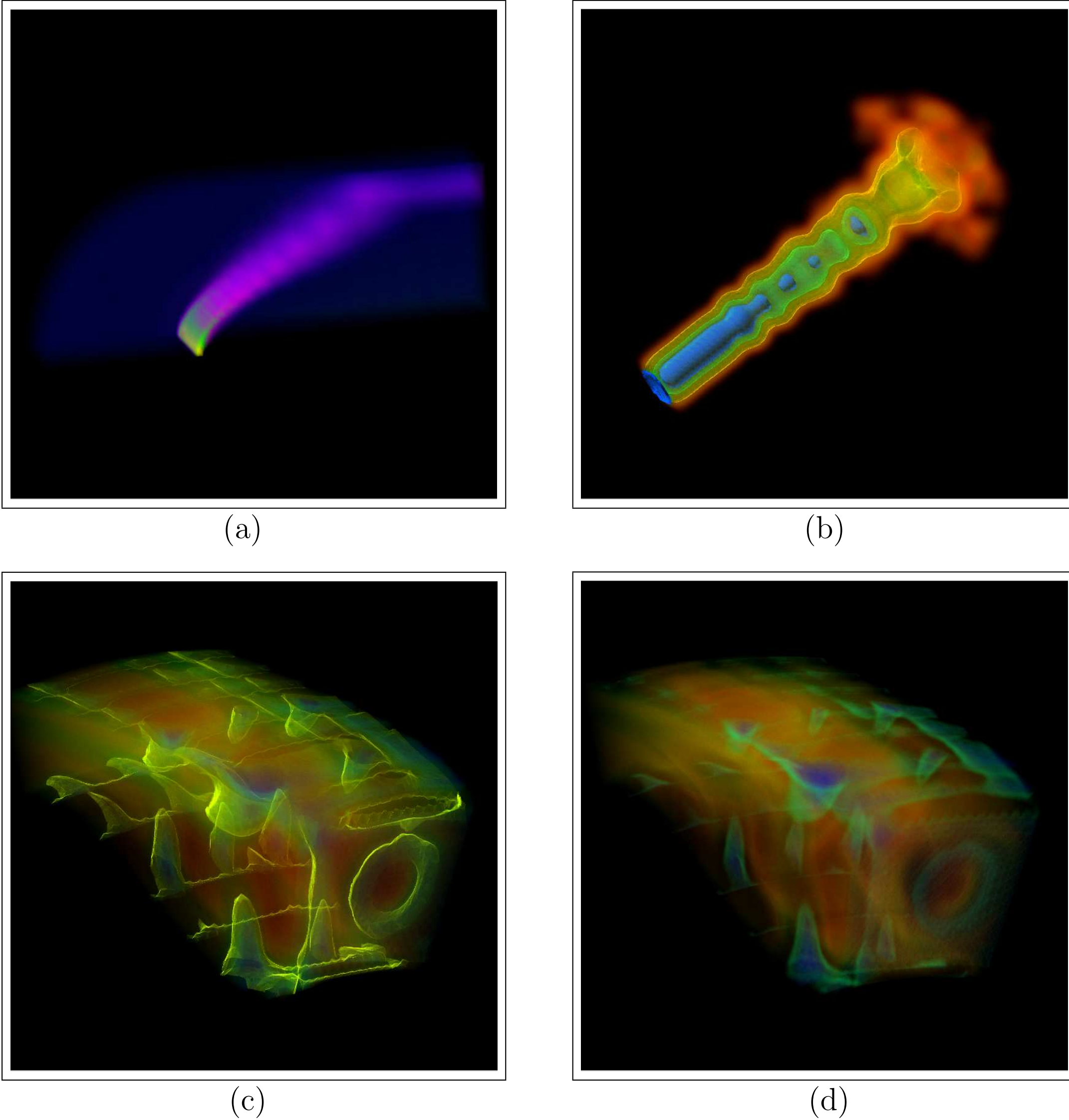


Figure 5.5: Images generated by IPTINT using the following datasets: (a) "blunt", (b) "fuel", and (c and d) "comb". All images were generated using partial pre-integration, and for the (b) "fuel" and (c) "comb" the iso-surfaces were also rendered.

In Tables 5.7 and 5.8 it is important to note the difference between two volumetric datasets tested. For the “blunt” dataset, our algorithm (IPTINT) is competitive with other algorithms even when doing a hybrid volume rendering by extracting iso-surfaces. However, for the “post” dataset, IPTINT loses for the ray-casting algorithms even if we consider the basic version without pre-integration and iso-surfaces. This characteristic can be attributed to the fact that, for some viewpoints, the model has small pixel area, while for cell-projection approaches this pixel area size is irrelevant.

The different renderings can be seen in Figures 5.3 and 5.4. The “spx+” dataset shown in Figure 5.3 was rendered with the basic average scalar method (equivalent to use the original PT algorithm). While, in Figure 5.4, the image of the (a) “post” dataset was generated applying only the partial pre-integration technique (equivalent to use the original PTINT algorithm), and, in case of (b) “neghip”, both pre-integration and illuminated iso-surfaces were used.

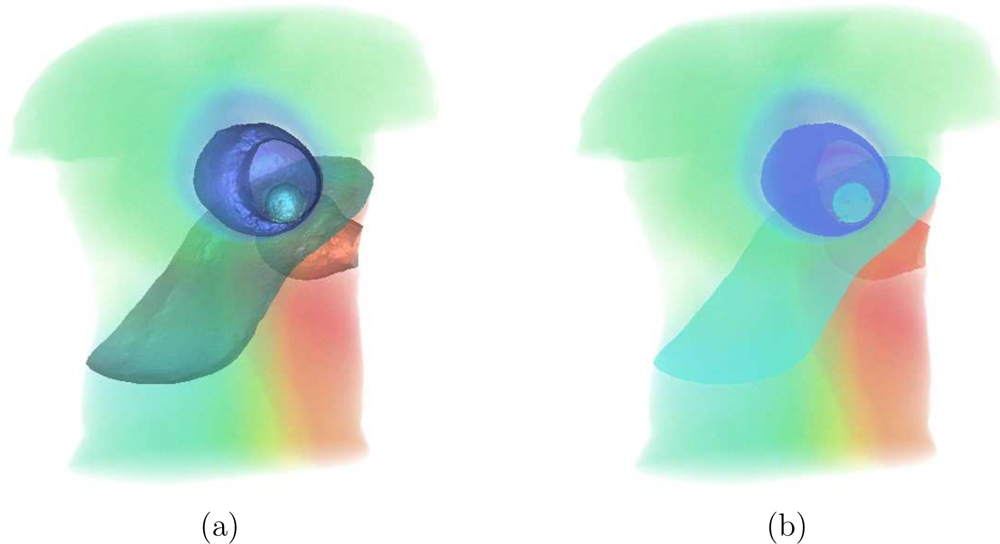
HAPT We tested the HAPT algorithm with the following irregular volume data: “blunt”; “post”; “spx+”; “delta”; “torso”; and “fighter”. Additionally, we use the “turbjet” dataset which varies over time to illustrate the flexibility of our algorithm. Figures 5.6, 5.7, 5.8 and 5.9 show some of these datasets. The dataset sizes and rendering timings are presented in Table 5.9. The timings are given using a viewport with 512 × 512 pixel resolution and considering the model always rotating. All measurements were performed on an Intel Xeon E5345 CPU with 2.33 GHz per processor and 4 GB of RAM, using a GeForce 8800 GTX with 768 MB of VRAM (Video RAM or GPU memory).

Dataset	Size		*DVR*		*ISO*		*DVR + ISO*	
	# Verts	# Tet	*fps*	*M Tet/s*	*fps*	*M Tet/s*	*fps*	*M Tet/s*
blunt	40 K	187 K	19.2	3.59	25.5	4.78	7.7	1.44
post	110 K	513 K	8.1	4.15	11.9	6.10	3.0	1.51
spx+	150 K	828 K	7.4	6.11	8.2	6.76	1.9	1.57
delta	211 K	1 M	4.5	4.52	6.0	6.01	1.5	1.51
torso	168 K	1.08 M	5.6	6.08	7.2	7.78	1.7	1.82
fighter	256 K	1.40 M	4.2	5.83	5.0	7.06	1.1	1.60
turbjet	212 K	1.01 M	17.5	17.67	n/a	n/a	n/a	n/a

Table 5.9: Dataset sizes and performance measurement of the HAPT algorithm applying the direct volume rendering technique (*DVR*), iso-surface rendering (*ISO*) or both.

For these results, we have used either direct volume rendering, iso-surface rendering, or both (see Figure 5.6 and 5.7). For the last two cases, we have fixed a maximum of four iso-surfaces. One issue is that the geometry shader requires the

specification of the maximum number of output vertices, which even when not fully used has an expressive impact on performance. This implies a significant overhead when rendering with both methods (last two columns of Table 5.9). Even though it is unlikely that all four iso-surfaces will cross one tetrahedron at the same time, we have accounted for this to be consistent with the results, but efficiency can be greatly improved without considerably limiting the algorithm.



(a) (b)

Figure 5.6: Volumetric data "torso" rendered with direct and indirect volume rendering, the iso-surfaces are rendered with (a) or without (b) illumination.

Table 5.10 compares our results with other GPU-based direct volume rendering algorithms (without extracting iso-surfaces nor having any illumination effect) using the "spx+". The sorting and drawing times are further detailed in the corresponding columns.

Algorithm	*Sort*	*Draw*	*fps*	*M Tet/s*
HAPT^Q	0.03 s	0.09 s	7.4	6.11
HAPT^B	0.04 s	0.09 s	6.9	5.73
HAPT^S	0.08 s	0.09 s	5.4	4.50
HAPT^M	0.13 s	0.09 s	4.4	3.61
HAVS^2	0.09 s	0.11 s	5.0	4.14
HAVS^6	0.09 s	0.12 s	4.7	3.94
PTINT	0.19 s	0.20 s	2.4	2.06
GATOR	0.08 s	0.83 s	1.1	0.93
HARC	n/a	0.22 s	4.6	3.82
HARC (INT)	n/a	0.28 s	3.5	2.90

Table 5.10: Performance comparison of direct volume rendering using the "spx+" model (828 K Tet) for different algorithms and criteria. Variations of HAPT depend on the sorting method used: **Q**uicksort; **B**itonic-sort; **S**TL-sort; and **M**PVONC. For HAVS, it is the size of its k-buffer: $k = \mathbf{2}$ and $k = \mathbf{6}$.

The original PTINT algorithm uses an approximate bucket sorting during rotation. Here we take timings only for the complete CPU sorting; and even though PTINT's sort is equivalent to the STL sort method used by GATOR and HAPT^S, PTINT still transfers data from the GPU for reordering during this step. Moreover, in both cases for HAVS, sorting time accounts only for the CPU pre-ordering, while for the HARC approaches there is no sorting step, since ray-casting algorithms traverse the entire volume using an adjacency data structure.

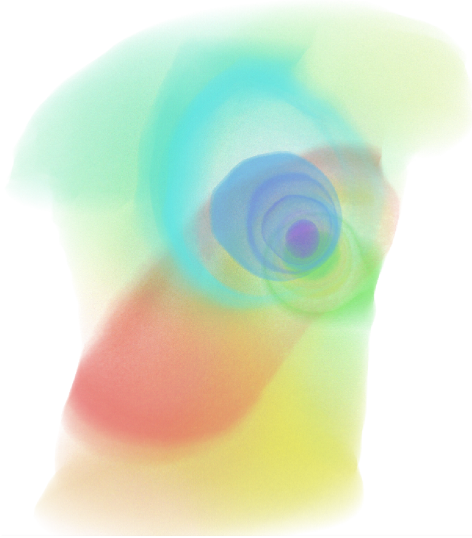


Figure 5.7: Direct volume rendering of the "torso" dataset (without iso-surface extraction).

An important remark is that HAPT has better rendering time than the compared algorithms, including cell-projection, ray-casting or hybrid approaches, and it does not store the volume on the GPU memory as done by PTINT and HARC. On the other hand, HAVS and GATOR stream the volume similar to our approach, but perform a multi-pass or redundant streaming. The main variation in HAPT's performance comes from the shape of the tetrahedra, which influences the probability of the cell falling in each of the projection cases, dictating the number of generated triangles.

Table 5.11 specifies frames and tetrahedra per second for the "torso" and "fighter" datasets using HAPT as well as for the other methods compared. All results have been measured for direct volume rendering, flushing the graphics pipeline every frame, and thus not profiting from continuous streaming. The volume streaming uses VBOs for HAPT, HAVS and GATOR (PTINT and HARC read the volume from texture and can not benefit from VBOs and continuous streaming). This comparison shows that HAPT is faster even when dealing with datasets with more than one million cells.

Algorithm	"torso" 1082 K Tet		"fighter" 1403 K Tet	
	fps	*M Tet/s*	*fps*	*M Tet/s*
$HAPT^Q$	5.6	6.08	4.2	5.83
$HAPT^B$	4.3	4.68	3.6	5.09
$HAPT^S$	3.9	4.25	2.9	4.10
$HAPT^M$	1.6	1.73	1.2	1.62
$HAVS^2$	3.7	4.01	2.9	4.12
$HAVS^6$	3.3	3.60	2.7	3.89
PTINT	1.3	1.47	0.9	1.31
GATOR	0.7	0.76	0.4	0.56
HARC	4.8	5.19	3.8	5.33
HARC (INT)	3.9	4.22	3.0	4.21

Table 5.11: Comparison of the HAPT algorithm and other approaches for the "torso" and "fighter" datasets.

One way to test the streaming performance of our algorithm for larger datasets (tens of millions of cells) is to render time-varying data as a sequence of static volumes. We tested HAPT using continuous streaming and an early discard test, explained in Chapter 3, to render the "turbjet" dataset (see Figure 5.8). This time-varying volume consists of 150 frames with 1 million cells per frame, thus the entire animation is composed of 150 million different cells (see Table 5.9 for more details). For this dataset size, algorithms that rely on storing the whole volume on the GPU memory, such as PTINT (and the improved version IPTINT presented in this thesis) and HARC, would require uploading and downloading different chunks of animation frames several times, and consequently would suffer a major performance loss.

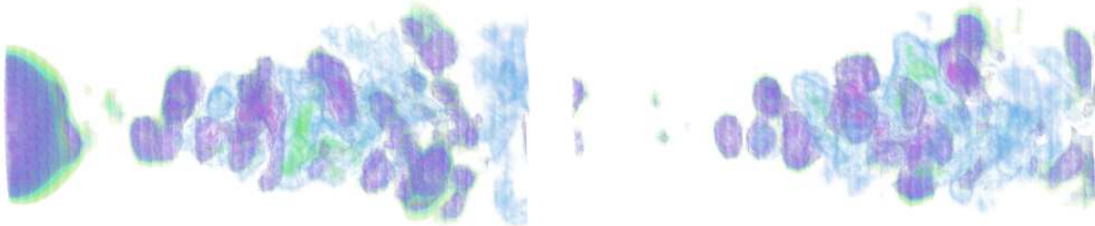
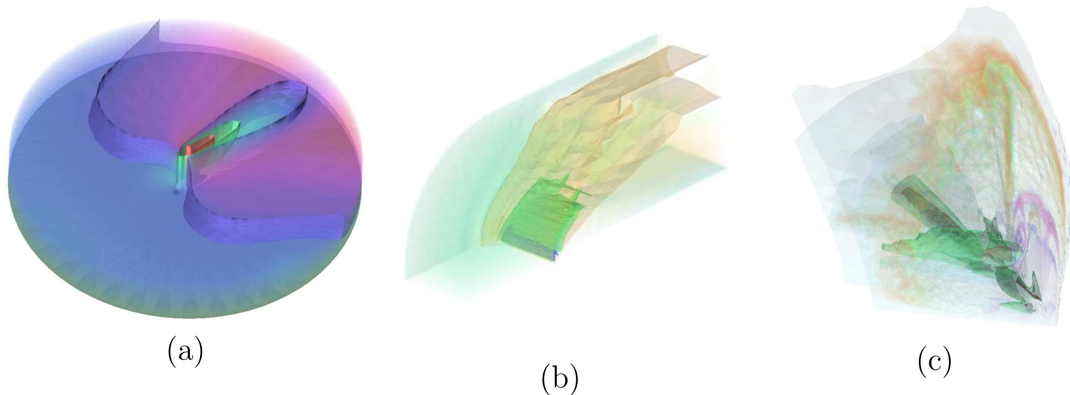


Figure 5.8: Two of the 150 frames from the time-varying volume "turbjet" (with 1 M Tet per frame) rendered using the HAPT algorithm. Changing HAPT's framework in the vertex shader, in order to perform an early discard test, HAPT was able to interactively render this sequence at 17 fps.

GATOR and PTINT are the most similar methods to HAPT since they also built upon the original PT algorithm. Nevertheless, both previous algorithms rely on strategies to trick the graphics card in rendering tetrahedra. In contrast, HAPT avoids the data redundancy when streaming the volume imposed by GATOR, and storing the dataset on the GPU memory as done by PTINT. Our approach also avoids PTINT's two-pass approach by rendering in a single pass through the graphics pipeline.

The usage of the partial pre-integration technique improves the volume rendering quality of GATOR and the original PT, placing our algorithm in the same quality category of PTINT. When compared with ray-casting approaches, such as HARC, and a finer interpolation scheme, such as HAVS, HAPT presents the rendering problems of interpolating scalar values and traversal length at extremities inherited from the PT idea. This problem is more pronounced when rendering regular volumes converted to tetrahedra, as done with the "turbjet" sequence shown in Figure 5.8.



(a) (b) (c)

Figure 5.9: Direct and indirect volume rendering with illuminated iso-surfaces of the following datasets: (a) "post"; (b) "blunt"; and (c) "fighter".

In terms of memory consumption, HAPT requires a fixed and very small amount of GPU memory to store the transfer function, classification table (the same TT Table used in PTINT) and the partial pre-integration table. Note that none of them are related to the volume size being rendered, yielding a total memory usage of about 10 KB. When streaming the volume using VBOs, our approach consumes additional GPU memory proportional to the dataset size. The memory-aware option is to send the primitives via streaming without using VBOs, losing 10% of the rendering performance, but avoiding memory limitations. This option of GPU memory usage by rendering primitives or applying VBOs can also be explored by GATOR and HAVS methods, since they also rely on streaming to render the volume. In contrast with these methods, HARC and HARC (INT) require the significant amount of 144 bytes/tet and 96 bytes/tet, respectively. Moreover, even an algorithm that has low

memory footprint, such as PTINT, would require 3 GB of GPU memory to fit the entire “turbjet” sequence.

Although different in nature, HAVS is still one of the most popular irregular volume renderers. One point in favor of HAVS is that it performs a more accurate visibility sorting, when compared with the three centroid sort methods of our approach (HAPT^Q, HAPT^B and HAPT^S). However, our implementation offers a flexible pipeline, has a gain of approximately 50% in frame rates in the case of $\text{HAPT}^Q \times \text{HAVS}^6$, and does not require multiple rendering passes. This latter characteristic may impact the overall rendering performance depending on the GPU.

In addition, our strategy totally decoupled sorting from rendering, which leads to several benefits over previous algorithms. PTINT relies on bringing the data back to the CPU to be sorted in between passes. HAVS has sorting embedded with rendering and relies on two different sorting passes, one on the CPU and one on the GPU. HARC, as any ray-casting method, does not require us to sort the volume cells, but on the other hand, it relies on auxiliary data structures that increase memory fetches and, consequently, decrease rendering performance. Moreover, the volume has to be loaded to the GPU texture memory, further limiting the volume size.

Comparison among the algorithms When we compare the volume rendering algorithms presented in this thesis, we see that VF-Ray-GPU does not present the rendering problems, inherited from the PT idea, when interpolating scalar values and traversal distance by the ray at the extremities of the tetrahedron projection. However, the PT-based approaches tend to be faster and cheaper in terms of memory consumption. RPTINT can handle only regular data, making it faster than the others when dealing with this type of volume. HAPT is the fastest irregular volume rendering algorithm, but it lacks iso-surface Phong shading presented in IPTINT. The HAPT algorithm generates the triangles of the iso-surfaces during volume rendering in the geometry shader, which makes the normal computation per vertex, required by Phong shading, impossible. On the other hand, IPTINT illuminates the iso-surfaces considering the gradient vectors of the scalar field as normals, allowing a better per-pixel illumination model.

5.2 Mesh Processing

The mesh processing method introduced in this thesis presents the concept of the dual space of a mesh, defining a novel similarity descriptor and how to use it to embed vertices in the mesh dual space. In addition to the use of this similarity space to augment mesh processing tasks, the Euclidean distances among points in this space can also reveal reflective symmetries. These symmetries can be viewed as a sub-set of the more general class of similarities described by our SAMPLE method. Figure 5.10 depicts the Stanford Armadillo model and its dual space with respect to one vertex. Note the symmetric patterns which appear naturally when coloring vertices by the distance order in the dual space.

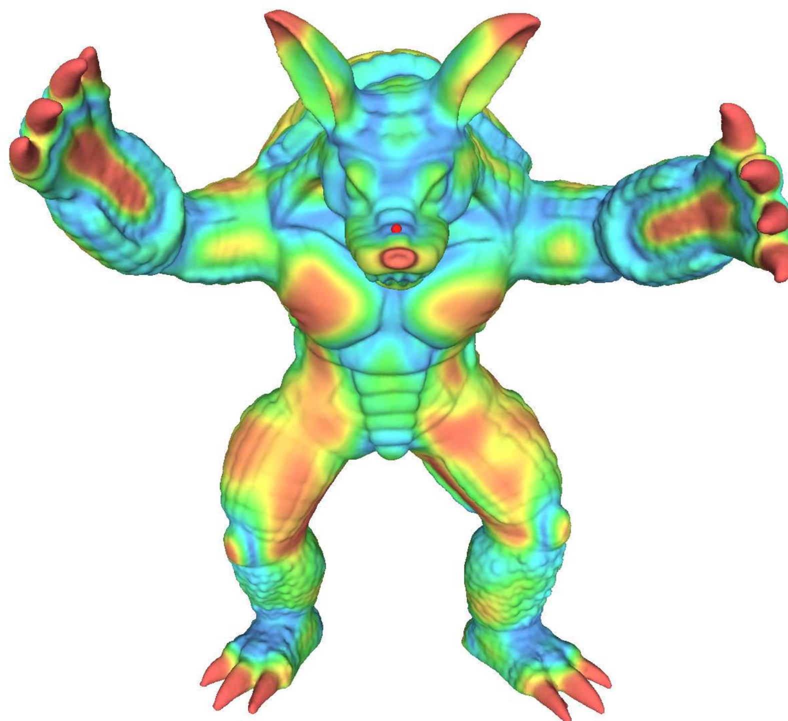


Figure 5.10: Dual space example for a given vertex (in red). The surrounding vertex region is used to map the entire mesh using the similarity neighborhood. Regions with wrinkles similar to the selected vertex on the nose are painted with blue, such as the ones in the arms and feet, while more dissimilar regions are gradually painted from green to red.

Another interesting result is the 3DScanCo Angel model, shown in Figure 5.11. Unlike the Armadillo, the Angel does not exhibit a simple plane of symmetry, yet it still contains an underlying symmetric structure. The dual space for this model also highlights the symmetric patterns when coloring the vertices by similarity distance.

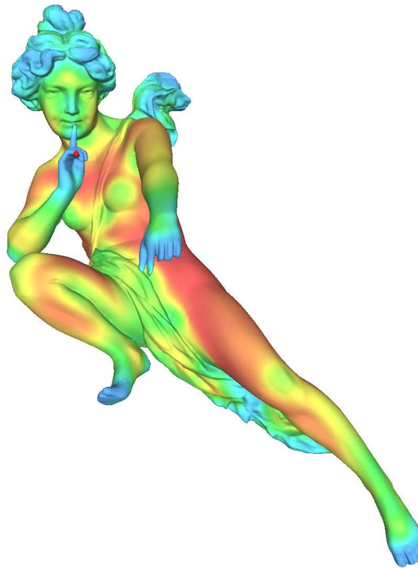


Figure 5.11: Symmetry example using the dual space. For a vertex on the right hand (in red), the model is painted from close (in blue) to far (in red) regions in the dual space.

While the dual space can reveal global symmetries, considering only k nearest neighbors in this space is an interesting special case. In particular, when $k = 1$, the dual space provides a result which closely matches one's intuition. Figure 5.12 illustrates how the mesh parameterization of one eye of the Armadillo model can be seamlessly duplicated to the other eye.

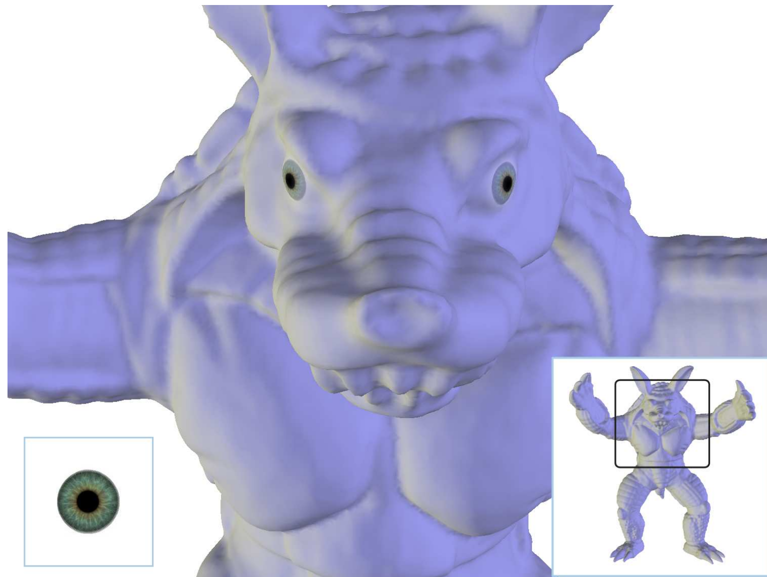


Figure 5.12: Application example of the immediate dual neighbor. The vertices in the center of each eye are immediate neighbors in dual space, choosing either vertex and applying our similarity augmented mesh parameterization leads to the automatic mapping of a texture (bottom left) to both eyes.

Analyzing the k nearest neighbors in similarity space is also interesting to reveal symmetric regions in high-detailed models. Take for example the *XYZ RGB Thai Statue* model shown in Figure 5.13. Even with a large number of details on the surface, our dual space is able to capture small similarities using $k = 0.5\%$ of the total number of vertices.

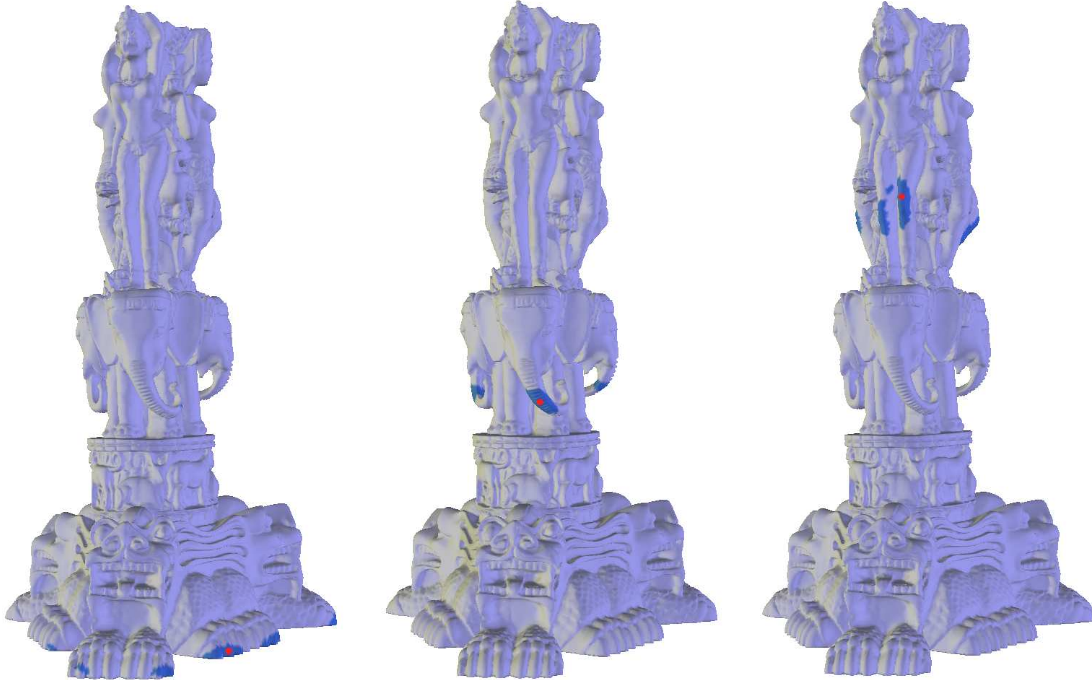


Figure 5.13: Nearest similarities (in blue) of the XYZ RGB Thai Statue model (with 125 K vertices). Three vertices (in red) are used in this example: on the bottom of the statue (left) finding several toes; finding the three elephants' trunk in the middle; and on the top (right) finding several knees.

The pre-computation time to build the dual space is shown in Table 5.12 and depends on the following computations: the heightmap descriptor, which is a standard procedure linear in the number of vertices; the expansion in Zernike coefficients of each heightmap image; and the computation of Gaussian-weighted Zernike coefficients, which is a fixed neighborhood summation also linear in the number of vertices. The computation of Zernike coefficients requires, for the XYZ RGB Asian Dragon model (with 108 K vertices), 5.54 seconds on a 2.33 GHz Intel Xeon E5345 CPU with 4 GB RAM. Moreover, the Stanford Armadillo (with 172 K vertices) and 3DScanCo Angel (with 237 K vertices) models require 9.01 and 12.15 seconds, respectively, providing empirical validation that the computation of the Zernike coefficients also scales linearly in the number of vertices.

Model	# Verts	# Faces	Heightmap	Zernike	Gaussian
Dama	106 K	207 K	3.20 min	5.53 sec	2.47 min
Asian Dragon	108 K	216 K	2.56 min	5.54 sec	1.83 min
Thai Statue	125 K	250 K	2.98 min	6.58 sec	2.07 min
Armadillo	172 K	345 K	6.12 min	9.01 sec	5.34 min
Angel	237 K	474 K	17.76 min	12.15 sec	18.45 min

Table 5.12: Computation time for establishing the dual space. The first two columns show the number of vertices and faces, while the last three columns show the time spent in each pre-computation step of our algorithm.

The run time to compute dual neighbors for a given vertex using the pre-computed dual space depends on the evaluation of similarity differences and nearest neighborhood searching. We apply a standard sorting method to the similarity differences, making the search trivial. The computation of the differences and the sorting operations for a selected vertex require 0.02 seconds for the XYZ RGB Asian Dragon model. The propagation time, on the other hand, mainly depends on the mesh processing task to propagate. In our experiments, the parameterization of one dragon scale consumed 0.01 seconds, while the propagation to the 57 similar scales consumed 0.26 seconds. Finally, the detail transfer applied to the same scales consumed 0.29 seconds.

Chapter 6

Conclusions

"Not only is the universe stranger than we imagine, it is stranger than we can imagine."
– Arthur Eddington

In this thesis we have presented efficient GPU approaches for volume rendering and introduced the idea of similarity augmented mesh processing using local exemplars. Each volume rendering approach aims to offer advantages on a specific context: VF-Ray-GPU is a memory-aware algorithm able to render large datasets using ray casting on the GPU; RPTINT specializes the original PT method to regular data using graphics hardware, greatly increasing performance; IPTINT adapts PT using a two-step approach on the GPU, adding indirect volume rendering through iso-surface detection and gradient interpolation; HAPT is efficient by avoiding multiple rendering passes, and flexible allowing easy mixing and matching of other strategies, such as sorting methods and rendering techniques. The former two algorithms were published in the following international conferences: VF-Ray-GPU in *Volume Graphics* 2008 [55]; and RPTINT in *GRAPP* 2007 [56]. The latter two algorithms were published in the international journal *Computer Graphics Forum*, IPTINT in 2008 [59] and HAPT in 2010 [63] as a special issue of the EuroVis 2010 conference. We make available source codes along with each paper in the hope that the research contributions of this thesis are as reproducible as possible.

The work presented on mesh processing was done during the CNPq's *sandwich* program (an internship international program) that I did with Professor Varshney at the University of Maryland in 2009. In this work, we defined a similarity measurement based on a heightmap image per vertex to be our surface descriptor in a new concept of dual space. The dual space is used to propagate mesh processing done on one region of a mesh to similar regions, using parameterization and editing

as two applications example. Our method, called SAMPLE, was recently accepted to be published in the Graphical Models journal [79].

We believe that our SAMPLE method shows promising results and presents numerous avenues for future research. One future direction of research is to consider different formulations for the local surface descriptor. The heightmap we currently use can be viewed as a scalar function parameterized over the tangent plane of the center vertex. It may be possible to form more informative descriptors by considering different parameterizations, such as *authalic* or *conformal*, for the region surrounding the center vertex. Another possibility is to consider different scalar functions, such as curvature or saliency, to be evaluated over the parameter domain. A third interesting avenue for future research is to extend our surface descriptor to take into account surface regions at multiple scales. The success of multi-scale descriptors in the field of image processing suggests that such an approach might have significant benefits over the current single-scale implementation of our SAMPLE method.

Appendix A

Developed Algorithms

- **VF-Ray-GPU Algorithm** [55] (*Visible-Face Driven Ray Casting implemented on the GPU*) – source code: http://code.google.com/p/vfray; doi: http://doi.ieeecomputersociety.org/10.2312/VG/VG-PBG08/155-162.
- **RPTINT Algorithm** [56] (*Regular Projected Tetrahedra with Partial Pre-Integration*) – source code: http://code.google.com/p/rptint; doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6833.
- **IPTINT Algorithm** [59] (*Improved Projected Tetrahedra with Partial Pre-Integration*) – source code: http://code.google.com/p/ptint; doi: http://dx.doi.org/10.1111/j.1467-8659.2007.01038.x.
- **HAPT Algorithm** [63] (*Hardware-Assisted Projected Tetrahedra*) – source code: http://code.google.com/p/hapt; doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01673.x.
- **SAMPLE Algorithm** [79] (*Similarity Augmented Mesh Processing using Local Exemplars*) – source code and publication are not available yet.

The volume rendering algorithms are summarized in the table below in terms of performance, memory consumption, treated data types (**I**rregular or **R**egular), integration quality and rendering types (**D**irect or **I**ndirect). The numbers 1 to 4 classify the algorithms on each corresponding aspect.

Algorithm	Performance	Memory	Data	Integration	Rendering
VF-Ray-GPU	4	1	I	3	D
RPTINT	1	3	R	4	D
IPTINT	3	4	I	2	D+I
HAPT	2	2	R+I	1	D+I

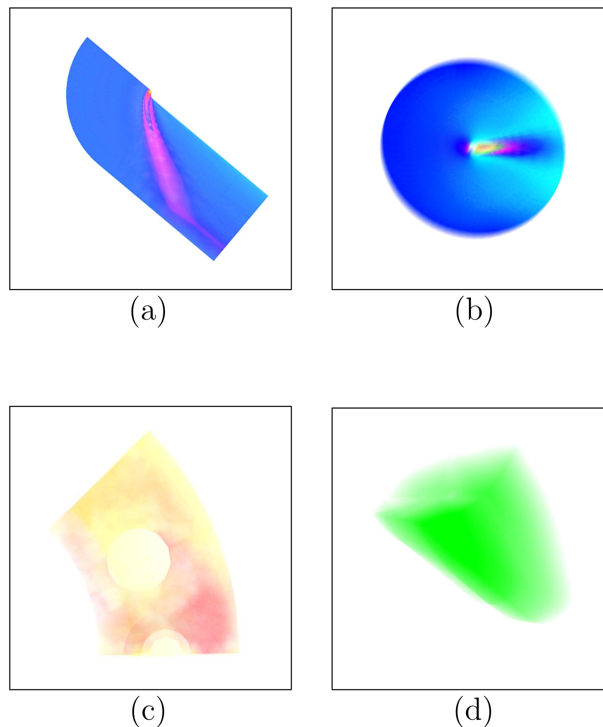
Appendix B

VF-Ray Algorithm

Visible-Face Driven Ray Casting [53] – http://code.google.com/p/vfray

The VF-Ray algorithm forms the basis of the VF-Ray-GPU algorithm presented in this thesis in Section 3.1. This appendix complements the explanation provided with figures and tables for VF-Ray.

The figure below shows four rendering results using VF-Ray: the *Blunt Fin* (a) and the *Oxygen Post* (b) are experiments on the interaction of oxygen in an environment; the *SPX* (c) presents areas of possible leaks of a reactor; and the *Delta Wing* (d) is an experiment with a delta wing:



(a) (b)

(c) (d)

The experiments performed in VF-Ray were conducted on an Intel Pentium IV 3.6 GHz with 2 GB of RAM and 1 MB of L2 cache memory. The algorithm was written in C/C++ ANSI without using any particular graphics library.

The following table shows results of memory consumption (in Mega Bytes), in rendering one frame at different resolutions, of the VF-Ray algorithm to the four volumes exhibited on the previous page. The percentages presented in this table are the memory consumption ratio of the other algorithms over VF-Ray. The following algorithms were tested: *ME-Ray – Memory Efficient Ray Casting* and *EME-Ray – Enhanced Memory Efficient Ray Casting* of PINA *et al.* [54]; *Bunyk et al.*'s algorithm [11]; and *ZSweep* of FARIAS *et al.* [80].

Volume	Resolution	Memory (MB)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512×512	14	404%	166%	528%	208%
	1024×1024	30	240%	129%	297%	177%
	2048×2048	39	333%	251%	377%	369%
	4096×4096	75	495%	451%	517%	689%
	8192×8192	219	610%	655%	617%	917%
Post	512×512	63	165%	74%	290%	97%
	1024×1024	66	203%	95%	301%	129%
	2048×2048	74	281%	172%	353%	238%
	4096×4096	110	424%	349%	470%	499%
	8192×8192	256	601%	560%	603%	800%
SPX	512×512	96	215%	74%	299%	87%
	1024×1024	99	234%	88%	305%	107%
	2048×2148	108	271%	141%	337%	188%
	4096×4096	144	383%	284%	428%	407%
	8192×8192	288	533%	498%	569%	711%
Delta	512×512	116	167%	74%	303%	97%
	1024×1024	119	200%	84%	308%	116%
	2048×2048	129	246%	124%	330%	177%
	4096×4096	165	346%	247%	403%	375%
	8192×8192	307	500%	467%	534%	704%

Next table shows results of performance (in seconds), in rendering one frame at different resolutions, of the VF-Ray algorithm and other volume rendering algorithms. The volumes used and algorithms compared are the same as the previous table.

Volume	Resolution	Time (s)	ME-Ray	EME-Ray	Bunyk	ZSweep
Blunt	512×512	1.9	139%	296%	105%	253%
	1024×1024	7.0	143%	317%	94%	431%
	2048×2048	27.2	146%	337%	88%	481%
	4096×4096	107.4	147%	328%	88%	516%
	8192×8192	426.7	154%	341%	91%	382%
Post	512×512	5.0	138%	251%	80%	201%
	1024×1024	19.5	136%	254%	76%	167%
	2048×2048	78.6	133%	258%	74%	194%
	4096×4096	309.9	135%	257%	74%	227%
	8192×8192	1246.3	135%	263%	72%	246%
SPX	512×512	4.1	111%	161%	76%	287%
	1024×1024	13.0	103%	203%	81%	202%
	2048×2048	46.6	103%	225%	83%	219%
	4096×4096	177.1	105%	234%	82%	226%
	8192×8192	696.3	105%	238%	81%	260%
Delta	512×512	3.1	130%	242%	91%	465%
	1024×1024	11.2	122%	270%	88%	271%
	2048×2048	41.9	121%	280%	87%	312%
	4096×4096	162.7	120%	290%	84%	341%
	8192×8192	640.3	123%	290%	83%	369%

The VF-Ray algorithm (*Visible-Face Driven Ray Casting*) was published under the title *Memory-Aware and Efficient Ray-Casting Algorithm* at the international conference SIBGRAPI 2007 [53].

Appendix C

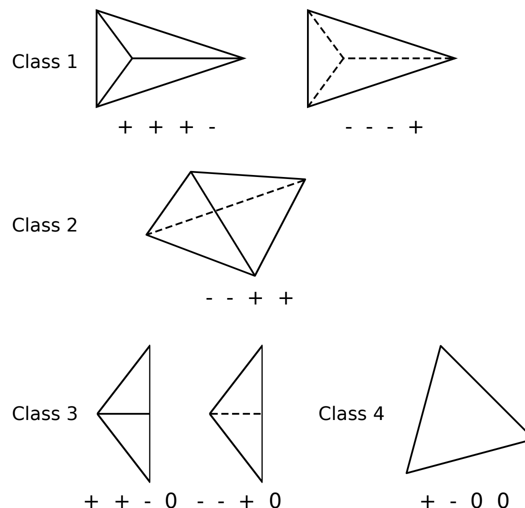
PT Algorithm

Projected Tetrahedra [6] – http://doi.acm.org/10.1145/99308.99322

The PT algorithm forms the main basis of the cell-projection algorithms presented in this thesis. This appendix explains the original PT algorithm of SHIRLEY and TUCHMAN [6].

The *Projected Tetrahedra* algorithm (PT) consists of projecting tetrahedra onto the image plane, and composing the projections in visibility ordering. In the case of volume cells different than tetrahedra, each cell must be divided into tetrahedra. In the original PT algorithm a method is presented to tetrahedralize a cube (or hexahedron) aiming to handle regular meshes. This idea is used in this thesis in the RPTINT algorithm presented in Section 3.2.

The shape of the projected tetrahedra is classified into four different classes, as shown in the figure below. The classes represent the four possible silhouettes of the tetrahedron projection into the image plane. The notation $+$, $-$ and 0 used is explained next. Note that, the classes 3 and 4 are degenerate cases of classes 1 and 2, where one of the vertices is projected over an edge (class 3) or over another vertex (class 4).

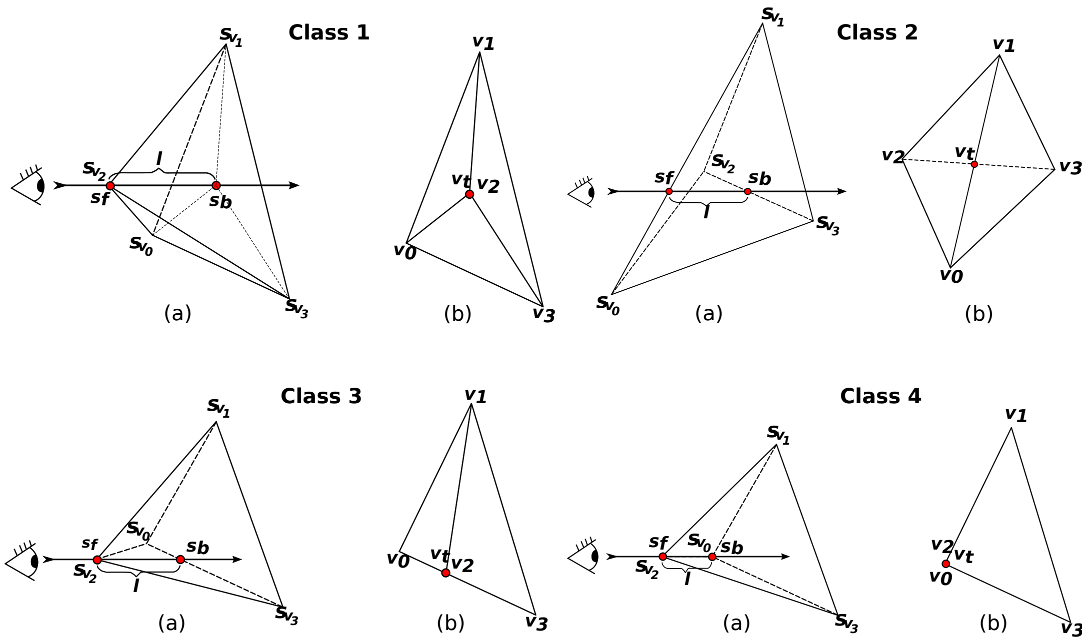


The cell classification is done based on the face normals of the tetrahedron. The cell is classified comparing the direction of the normal against the viewing vector. The faces are marked with the notation shown on the previous page, depending on whether:

- The normal points in the direction of the viewpoint: $+$;
- The normal points in the opposite direction of the viewpoint: $-$;
- The normal is perpendicular to the viewing vector: 0.

For each projected tetrahedron, the *thick vertex* is defined as the projection of the entry and exit points of the ray segment that traverses the maximum distance inside the tetrahedron. The size of this segment is defined as the *cell thickness*. All the other projected vertices are called *thin vertices*, since rays passing through theses vertices traverse zero distance inside the tetrahedron.

The figure below illustrates one case of each projection class. The tetrahedron (a) is shown projected onto the image plane (b) for each case. The vertices v_i are the projections of the original vertices of the tetrahedron, where s_{v_i} is the scalar value of the vertex. The thick vertex is defined as v_t and its attributes are: the cell thickness l; the front s_f and back s_b scalar values (entry and exit points).



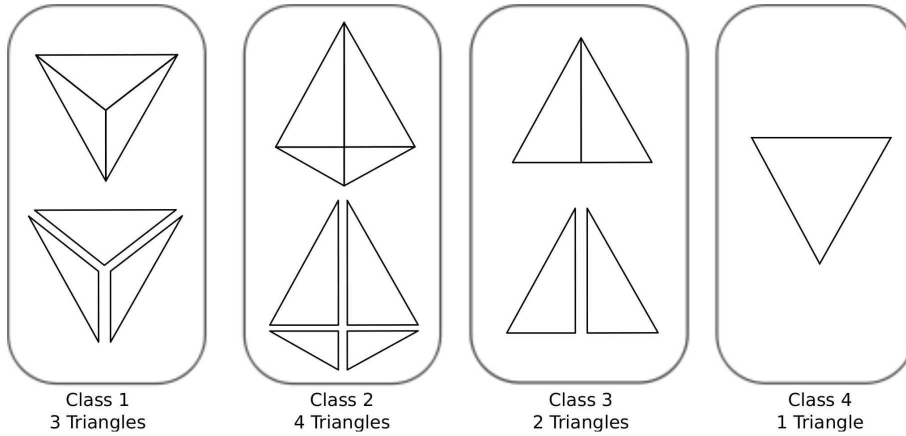
Analyzing the figure above, for the class 4 projection, the thick vertex v_t is defined as v_2 or v_0, since these two vertices are colinear with respect to the viewing vector. Consequently, $s_f = s_{v_2}$, $s_b = s_{v_0}$ and l is equal to the edge length of v_2 and v_0 in the original non-projected tetrahedron.

For other classes it is necessary to compute the attributes of the thick vertex v_t. The thickness l is computed by performing the inverse projection of v_t, finding the entry and exit points, and computing the Euclidean distance between these points. In the case of class 1, a bilinear interpolation should be performed:

$$v_t = v_0 + u(v_1 - v_0) + t(v_3 - v_0). \tag{C.1}$$

The (x, y) coordinates of v_t are given by the projection. In the PT algorithm, the Equation above is used to compute the z coordinate of v_t. For example, in the class 1 projection shown in the previous page, the thickness l is computed by the distance between the original tetrahedron vertex v_2 and the interpolation of vertices v_0, v_1 and v_3. The scalars front and back, s_f and s_b, of the thick vertex v_t are computed by interpolating the scalars of the original tetrahedron.

Each tetrahedron is decomposed into 1, 2, 3 or 4 triangles. The number of triangles depends on the projection class, as shown in the next figure.



As discussed in Chapter 2, the color C and opacity α of the fragments are interpolated, once computed from the values s_f, s_b and l of the triangles' vertices, according to the Equations 2.2 and 2.3. The fragment composition of the rendered triangles is performed following the rules of Equation 2.4 and 2.5.

The cell-projection algorithms presented in this thesis use the idea of the PT algorithm. In the next appendix, the base adaptation of the PT algorithm to the GPU will be presented, the PTINT algorithm [25].

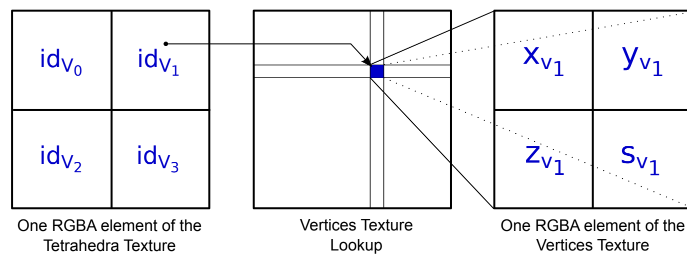
Appendix D

PTINT Algorithm

PT with Partial Pre-Integration [25] – http://code.google.com/p/ptint

The PTINT algorithm forms the basis of the RPTINT and IPTINT algorithms presented in this thesis in Sections 3.2 and 3.3. This appendix complements the explanation provided with figures and tables for PTINT.

To access the vertices' coordinates of the tetrahedra of a given volume inside the GPU, the PTINT algorithm uses two textures: *Tetrahedra* and *Vertices*. The access to the Vertices Texture is done using the Tetrahedra Texture as follows:



To classify the projection of a tetrahedron, the PTINT algorithm uses four cross-product tests (shown below). The GLSL [2] function *sign* is similar to the notation employed by PT and returns −1, 0 or 1, depending on whether the argument is less than, equal to or greater than zero, respectively.

$$\begin{aligned} vec_1 &= v_1 - v_0 \\ vec_2 &= v_2 - v_0 \\ vec_3 &= v_3 - v_0 \\ vec_4 &= v_1 - v_2 \\ vec_5 &= v_1 - v_3 \end{aligned}$$

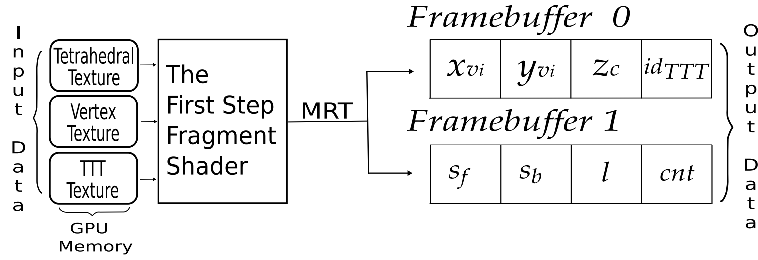
$$\begin{aligned} test_1 &= sign((vec_1 \times vec_2).z) + 1 \\ test_2 &= sign((vec_1 \times vec_3).z) + 1 \\ test_3 &= sign((vec_2 \times vec_3).z) + 1 \\ test_4 &= sign((vec_4 \times vec_5).z) + 1 \end{aligned}$$

After the classification tests, the PTINT algorithms uses the *Ternary Truth Table* (TTT), presented below, to determine the case of the projection class.

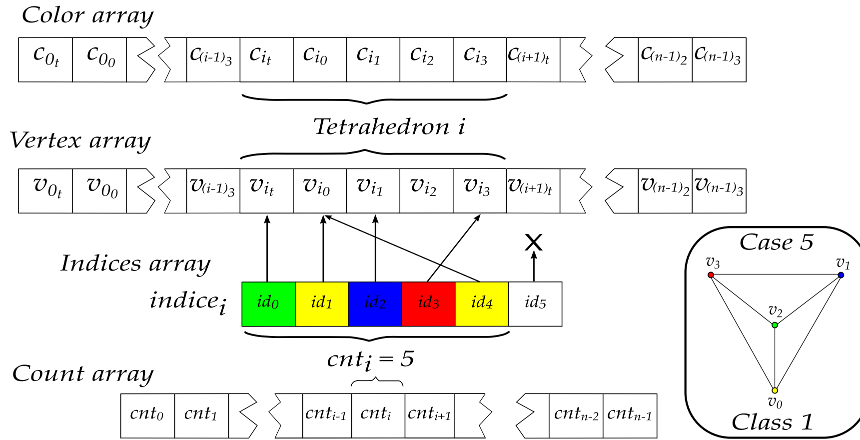
id_{ttt}	$test_{1234}$	$Case$	$RGBA$	id_{ttt}	$test_{1234}$	$Case$	$RGBA$
0	0 0 0 0	12	0-3-2-1	41	1 1 1 2	–	–
1	0 0 0 1	18	0-3-2-1	42	1 1 2 0	–	–
2	0 0 0 2	6	0-3-2-1	43	1 1 2 1	–	–
3	0 0 1 0	25	1-0-3-2	44	1 1 2 2	49	0-1-2-3
4	0 0 1 1	40	2-3-1-0	45	1 2 0 0	34	3-2-0-1
5	0 0 1 2	23	1-0-2-3	46	1 2 0 1	–	–
6	0 0 2 0	8	0-2-3-1	47	1 2 0 2	–	–
7	0 0 2 1	20	0-2-3-1	48	1 2 1 0	47	0-2-1-3
8	0 0 2 2	14	0-2-3-1	49	1 2 1 1	–	–
9	0 1 0 0	27	2-1-0-3	50	1 2 1 2	–	–
10	0 1 0 1	–	–	51	1 2 2 0	37	3-0-2-1
11	0 1 0 2	–	–	52	1 2 2 1	44	1-2-3-0
12	0 1 1 0	46	0-3-2-1	53	1 2 2 2	35	3-0-1-2
13	0 1 1 1	–	–	54	2 0 0 0	4	0-3-1-2
14	0 1 1 2	–	–	55	2 0 0 1	16	0-3-1-2
15	0 1 2 0	32	2-1-3-0	56	2 0 0 2	10	–
16	0 1 2 1	41	1-3-0-2	57	2 0 1 0	–	–
17	0 1 2 2	30	2-3-1-0	58	2 0 1 1	–	1-2-0-3
18	0 2 0 0	2	1-3-0-2	59	2 0 1 2	21	–
19	0 2 0 1	–	–	60	2 0 2 0	–	–
20	0 2 0 2	–	–	61	2 0 2 1	–	–
21	0 2 1 0	22	1-3-0-2	62	2 0 2 2	1	1-2-0-3
22	0 2 1 1	–	–	63	2 1 0 0	29	2-0-1-3
23	0 2 1 2	–	–	64	2 1 0 1	42	1-3-2-0
24	0 2 2 0	9	0-2-1-3	65	2 1 0 2	31	2-0-3-1
25	0 2 2 1	15	0-2-1-3	66	2 1 1 0	–	–
26	0 2 2 2	3	0-2-1-3	67	2 1 1 1	–	–
27	1 0 0 0	36	3-2-1-0	68	2 1 1 2	45	0-3-1-2
28	1 0 0 1	43	1-2-0-3	69	2 1 2 0	–	–
29	1 0 0 2	38	3-1-2-0	70	2 1 2 1	–	–
30	1 0 1 0	–	–	71	2 1 2 2	28	2-3-0-1
31	1 0 1 1	–	1-3-0-2	72	2 2 0 0	13	0-1-3-2
32	1 0 1 2	48	–	73	2 2 0 1	19	0-1-3-2
33	1 0 2 0	–	–	74	2 2 0 2	7	0-1-3-2
34	1 0 2 1	–	0-2-1-3	75	2 2 1 0	24	1-3-2-0
35	1 0 2 2	33	0-2-1-3	76	2 2 1 1	39	2-3-0-1
36	1 1 0 0	50	0-2-1-3	77	2 2 1 2	26	1-2-3-0
37	1 1 0 1	–	3-2-1-0	78	2 2 2 0	5	0-1-2-3
38	1 1 0 2	–	1-2-0-3	79	2 2 2 1	17	0-1-2-3
39	1 1 1 0	–	3-1-2-0	80	2 2 2 2	11	0-1-2-3
40	1 1 1 1	–	–	–	–	–	–

In the above table, the values 0, 1 and 2 for the columns labeled $test_{1234}$ are all the 81 possible results of the 4 classification tests presented in the previous page. These results form the index id_{TTT} of the TT Table.

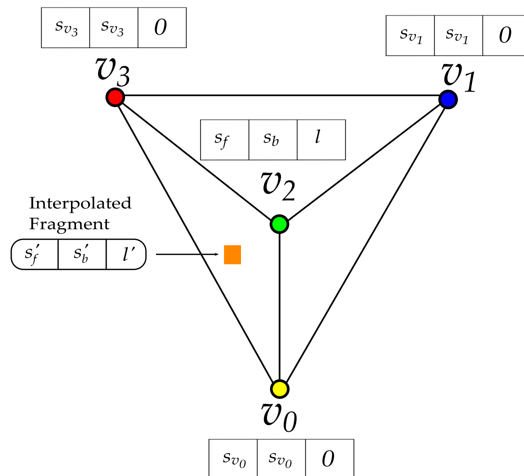
The first step of the PTINT algorithm is performed using the fragment shader with the following input/output scheme:



The main data structure of PTINT is based on vertex arrays to the *glMultiDrawElements* function of OpenGL [1]. In the figure below, the indices illustrate the case 5 (of class 1), where the correct order to render the tetrahedron i is $v_{i_t} - v_{i_0} - v_{i_1} - v_{i_3} - v_{i_0}$. Note that v_{i_2} is the thick vertex and its coordinates are copied between the first and second step of the algorithm to v_{i_t}.



The figure below illustrates an input fragment of the second step of the PTINT algorithm. The interpolated values use the class 1 example shown in previous figure. Note that, except for the thick vertex, all other vertices are rendered with the original values of the volume: s_{v_i}.

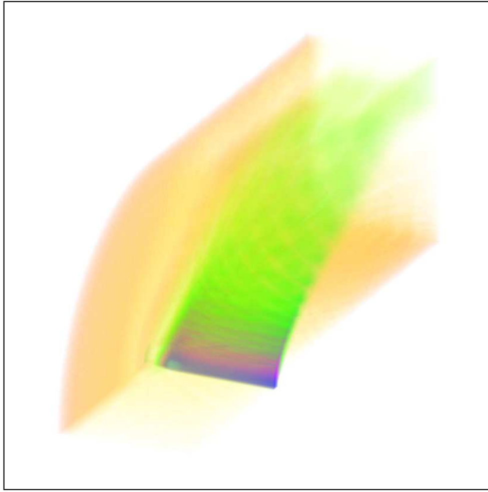


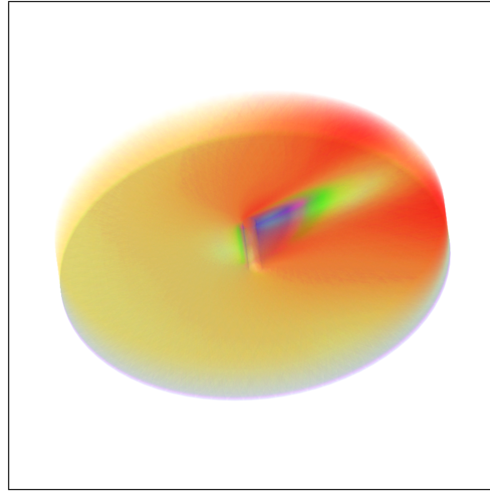
In the subsequent pages we show the figures and performance measures of the PTINT algorithm. The measurements were conducted on an Intel Pentium IV 3.6 GHz with 2 GB of RAM, using the nVidia GeForce 6800 graphics card with 256 MB and PCI Express 16x bus interface. The implementation of the PTINT algorithm was written in C/C++ using OpenGL 2.0 with GLSL under Linux.

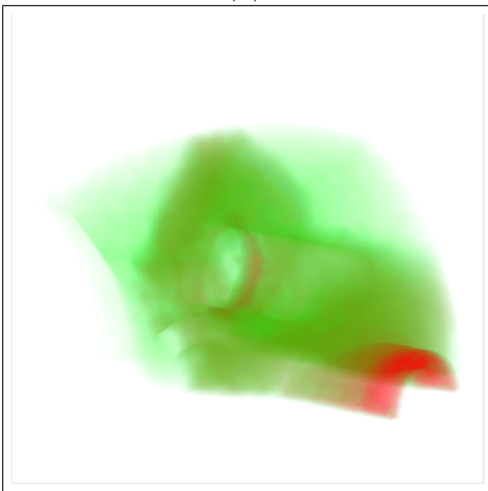
In the subsequent figures rendering examples are shown of six volumetric data using the PTINT algorithm:

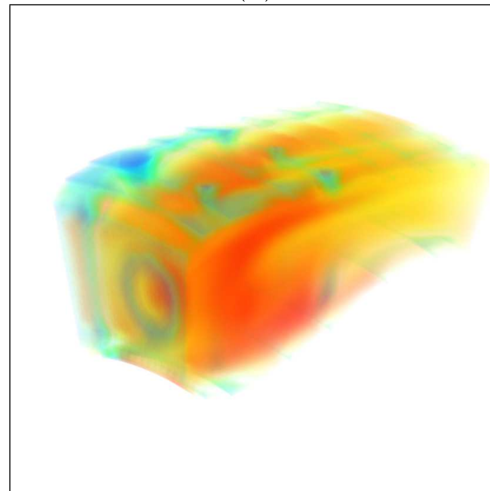
- the *Blunt Fin* (a) and the *Oxygen Post* (b) are experiences on the interaction of oxygen in an environment;
- the *SPX* (c) presents areas of possible leaks of a reactor, while the *Combustion Chamber* (d) shows different temperatures in a combustion chamber;
- the *Fuel Injection* (e) simulates the injection of fuel in a combustion chamber;
- and the *Brain Tomography* (f) is the result of a CT scan of a brain.

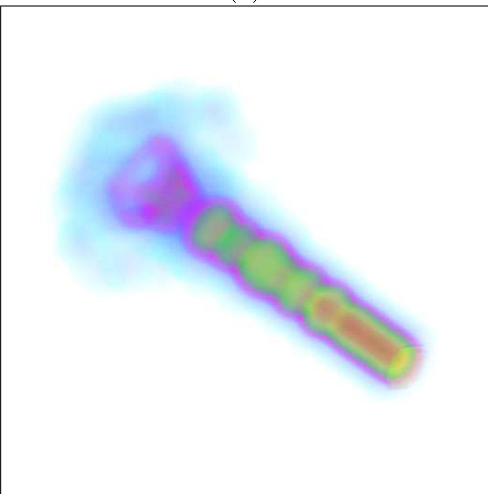
In the subsequent tables, results involving these six volumes are presented. The results aim to compare the PTINT algorithm with other state-of-art algorithms, based on cell projection or ray casting.

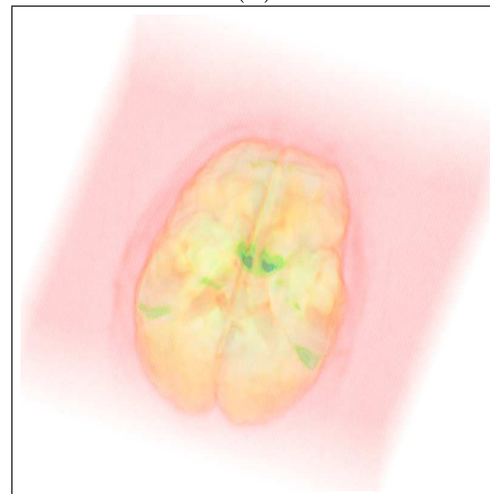

(a)


(b)


(c)


(d)


(e)


(f)

The following table presents performance comparison of the PTINT algorithm with other volume rendering algorithms. The following cell-projection and ray-casting algorithms were tested: *PTINT* – presented algorithm [57]; *GATOR – GPU Accelerated Tetrahedra Renderer* [9]; *VICP – View-Independent Cell Projection* (implemented on the GPU and on the CPU) [17]; *VICP (Balanced)* – VICP balanced [12]; *HARC – Hardware-Based Ray Casting* [22]; *HARC (INT) – HARC* with partial pre-integration [23]; *HAVIS – Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection* [8].

Algorithm / Dataset	Blunt Fin	Oxygen Post
PTINT	10.76 fps	4.35 fps
GATOR	4.07 fps	1.51 fps
VICP (GPU)	5.20 fps	1.93 fps
VICP (CPU)	1.82 fps	0.57 fps
VICP (Balanced)	4.10 fps	1.11 fps
HARC	4.47 fps	8.63 fps
HARC (INT)	4.94 fps	5.93 fps
HAVIS	2.36 fps	0.79 fps

The next table shows some results of the PTINT algorithm for different volumetric data. The timings consider the volume in constant rotation, and the image plane used for visualization has 512×512 pixels.

Dataset	Vertices	Tetrahedra	fps	M Tets/s
Blunt Fin	40 K	187 K	11.30	2.1197
Oxygen Post	110 K	513 K	4.49	2.3844
SPX	150 K	828 K	3.04	2.5269
Combustion Chamber	47 K	215 K	9.32	2.0054
Fuel Injection	262 K	1.5 M	1.49	2.2460
Brain Tomography	950 K	5.5 M	0.46	2.5608

The PTINT algorithm (*Projected Tetrahedra with Partial Pre-Integration*) was published under the title *GPU-Based Cell Projection for Interactive Volume Rendering* in the international conference SIBGRAPI 2006 [25].

Bibliography

[1] SHREINER, D., WOO, M., NEIDER, J., DAVIS, T., OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL(R) Programming Guide (Red Book)*. 7th ed. London, UK, Addison-Wesley Professional, 2009. ISBN: 978-0321552624. Available at: <`http://www.opengl.org/documentation/red_book/`>.

[2] ROST, R. J., LICEA KANE, B., GINSBURG, D., KESSENICH, J. M., LICHTENBELT, B., MALAN, H., WEIBLEN, M. *OpenGL(R) Shading Language (Orange Book)*. 3rd ed. London, UK, Addison-Wesley Professional, 2009. ISBN: 978-0321637635. Available at: <`http://www.3dshaders.com/home/`>.

[3] MARROQUIM, R., MAXIMO, A. "Introduction to GPU Programming with GLSL", *Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, pp. 3–16, 2009, doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI-Tutorials.2009.9.

[4] "CUDA Environment – Compute Unified Device Architecture". . nVidia CUDA, 2007. Available at: <`http://www.nvidia.com/object/cuda_home.html`>.

[5] MAX, N. "Optical Models for Direct Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, v. 1, n. 2, pp. 99–108, 1995. ISSN: 1077-2626, doi: http://dx.doi.org/10.1109/2945.468400.

[6] SHIRLEY, P., TUCHMAN, A. "A Polygonal Approximation to Direct Scalar Volume Rendering", *SIGGRAPH Comput. Graph.*, v. 24, n. 5, pp. 63–70, 1990. ISSN: 0097-8930, doi: http://doi.acm.org/10.1145/99308.99322.

[7] LEVOY, M. "Efficient Ray Tracing of Volume Data", *ACM Transactions on Graphics*, v. 9, n. 3, pp. 245–261, 1990. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/78964.78965.

[8] RÖTTGER, S., KRAUS, M., ERTL, T. "Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection". In: *VIS '00: Proceedings*

of the conference on Visualization '00, pp. 109–116, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press. ISBN: 1-58113-309-X, doi: http://doi.ieeecomputersociety.org/10.1109/VISUAL.2000.885683.

[9] WYLIE, B., MORELAND, K., FISK, L. A., CROSSNO, P. "Tetrahedral Projection Using Vertex Shaders", *Volume Visualization and Graphics, IEEE Symposium on*, pp. 7–12, 2002, doi: http://doi.ieeecomputersociety.org/10.1109/SWG.2002.1226504.

[10] UPSON, C., KEELER, M. "V-Buffer: Visible Volume Rendering". In: *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 59–64, New York, NY, USA, 1988. ACM Press. ISBN: 0-89791-275-6, doi: http://doi.acm.org/10.1145/54852.378482.

[11] BUNYK, P., KAUFMAN, A. E., SILVA, C. T. "Simple, Fast, and Robust Ray Casting of Irregular Grids". In: *Dagstuhl '97, Scientific Visualization*, pp. 30–36, Washington, DC, USA, 1997. IEEE Computer Society. ISBN: 0-7695-0505-8, doi: http://doi.ieeecomputersociety.org/10.1109/DAGSTUHL.1997.10002.

[12] MORELAND, K., ANGEL, E. "A Fast High Accuracy Volume Renderer for Unstructured Data". In: *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics*, pp. 13–22, Piscataway, NJ, USA, 2004. IEEE Press. ISBN: 0-7803-7641-2, doi: http://dx.doi.org/10.1109/VV.2004.2.

[13] MORELAND, K. D. *Fast High Accuracy Volume Rendering*. Doctoral Thesis, University of New Mexico, July 2004. Available at: <`http://www.cs.unm.edu/~kmorel/`>.

[14] SILVA, C. T., MITCHELL, J. S. B., WILLIAMS, P. L. "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes". In: *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pp. 87–94, New York, NY, USA, 1998. ACM. ISBN: 1-58113-105-4, doi: http://doi.acm.org/10.1145/288126.288170.

[15] KRAUS, M., ERTL, T. "Cell-Projection of Cyclic Meshes". In: *VIS '01: Proceedings of the conference on Visualization '01*, pp. 215–222, Washington, DC, USA, 2001. IEEE Computer Society. ISBN: 0-7803-7200-X, doi: http://doi.ieeecomputersociety.org/10.1109/VISUAL.2001.964514.

[16] COOK, R., MAX, N., SILVA, C. T., WILLIAMS, P. L. "Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data", *IEEE Transactions on Visualization and Computer Graphics*, v. 10, n. 6, pp. 695–707, Nov.-Dec. 2004. ISSN: 1077-2626, doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2004.45.

[17] WEILER, M., KRAUS, M., ERTL, T. "Hardware-Based View-Independent Cell Projection", *Volume Visualization and Graphics, IEEE Symposium on*, pp. 13–22, 2002, doi: http://doi.ieeecomputersociety.org/10.1109/SWG.2002.1226505.

[18] CALLAHAN, S., IKITS, M., COMBA, J., SILVA, C. "Hardware-Assisted Visibility Ordering for Unstructured Volume Rendering", *IEEE Transactions on Visualization and Computer Graphics*, v. 11, n. 3, pp. 285–295, 2005, doi: http://dx.doi.org/10.1109/TVCG.2005.46.

[19] BLINN, J. F. "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces". In: *SIGGRAPH '82: Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 21–29, New York, NY, USA, 1982. ACM Press. ISBN: 0-89791-076-1, doi: http://doi.acm.org/10.1145/800064.801255.

[20] LEVOY, M. "A Hybrid Ray Tracer for Rendering Polygon and Volume Data", *Computer Graphics and Applications, IEEE*, v. 10, n. 2, pp. 33–40, Mar 1990. ISSN: 0272-1716, doi: http://dx.doi.org/10.1109/38.50671.

[21] GARRITY, M. P. "Raytracing Irregular Volume Data". In: *VVS '90: Proceedings of the 1990 Workshop on Volume Visualization*, pp. 35–40, New York, NY, USA, 1990. ACM Press. ISBN: 0-89791-417-1, doi: http://doi.acm.org/10.1145/99307.99316.

[22] WEILER, M., KRAUS, M., MERZ, M., ERTL, T. "Hardware-Based Ray Casting for Tetrahedral Meshes". In: *VIS '03: Proceedings of the 14th IEEE Conference on Visualization*, pp. 333–340, 2003. ISBN: 0-7695-2030-8/03, doi: http://dx.doi.org/10.1109/VISUAL.2003.1250390.

[23] ESPINHA, R., CELES, W. "High-Quality Hardware-Based Ray-Casting Volume Rendering Using Partial Pre-Integration". In: *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, p. 273. IEEE Computer Society, 2005. ISBN: 0-7695-2389-7, doi: http://dx.doi.org/10.1109/SIBGRAPI.2005.29.

[24] KRAUS, M., QIAO, W., EBERT, D. S. "Projecting Tetrahedra Without Rendering Artifacts". In: *VIS '04: Proceedings of the 15th IEEE conference on Visualization '04*, pp. 27–34, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7803-8788-0, doi: http://dx.doi.org/10.1109/VIS.2004.85.

[25] MARROQUIM, R., MAXIMO, A., FARIAS, R., ESPERANÇA, C. "GPU-Based Cell Projection for Interactive Volume Rendering", *Computer Graphics and Image Processing, Brazilian Symposium on*, pp. 147–154, 2006. ISSN: 1530-1834, doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI.2006.22. Best Paper Award.

[26] KAZHDAN, M., CHAZELLE, B., DOBKIN, D., FUNKHOUSER, T., RUSINKIEWICZ, S. "A Reflective Symmetry Descriptor for 3D Models", *Algorithmica*, v. 38, n. 1, pp. 201–225, 2003. ISSN: 0178-4617, doi: http://dx.doi.org/10.1007/s00453-003-1050-5.

[27] PODOLAK, J., SHILANE, P., GOLOVINSKIY, A., RUSINKIEWICZ, S., FUNKHOUSER, T. "A Planar-Reflective Symmetry Transform for 3D Shapes". In: *SIGGRAPH '06: Proceedings of the 33th annual conference on Computer graphics and interactive techniques*, pp. 549–559, New York, NY, USA, 2006. ACM. ISBN: 1-59593-364-6, doi: http://doi.acm.org/10.1145/1179352.1141923.

[28] XU, K., ZHANG, H., TAGLIASACCHI, A., LIU, L., LI, G., MENG, M., XIONG, Y. "Partial Intrinsic Reflectional Symmetry of 3D Shapes". In: *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–10, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-858-2, doi: http://doi.acm.org/10.1145/1661412.1618484.

[29] RUSTAMOV, R. M. "Laplace-Beltrami eigenfunctions for deformation invariant shape representation". In: *SGP '07: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 225–233, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN: 978-3-905673-46-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6567.

[30] OVSJANIKOV, M., SUN, J., GUIBAS, L. J. "Global Intrinsic Symmetries of Shapes." *Comput. Graph. Forum*, v. 27, n. 5, pp. 1341–1348, 2008, doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01273.x.

[31] SUN, J., OVSJANIKOV, M., GUIBAS, L. J. "A Concise and Provably Informative Multi-Scale Signature Based on Heat Diffusion", *Comput. Graph. Forum*, v. 28, n. 5, pp. 1383–1392, 2009, doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01515.x.

[32] GOLOVINSKIY, A., PODOLAK, J., FUNKHOUSER, T. "Symmetry-Aware Mesh Processing". In: *Proceedings of the 13th IMA International Conference on Mathematics of Surfaces*, pp. 170–188, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN: 978-3-642-03595-1, doi: http://dx.doi.org/10.1007/978-3-642-03596-8_10.

[33] ZELINKA, S., GARLAND, M. "Similarity-Based Surface Modelling using Geodesic Fans". In: *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pp. 204–213, New York, NY, USA, 2004. ACM. ISBN: 3-905673-13-4, doi: http://doi.acm.org/10.1145/1057432.1057460.

[34] GAL, R., COHEN OR, D. "Salient Geometric Features for Partial Shape Matching and Similarity", *ACM Trans. Graph.*, v. 25, n. 1, pp. 130–150, 2006. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1122501.1122507.

[35] PODOLAK, J., GOLOVINSKIY, A., RUSINKIEWICZ, S. "Symmetry-Enhanced Remeshing of Surfaces". In: *SGP '07: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 235–242, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN: 978-3-905673-46-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7217.

[36] PALACIOS, J., ZHANG, E. "Rotational symmetry field design on surfaces", *ACM Trans. Graph.*, v. 26, n. 3, pp. 55, 2007. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1276377.1276446.

[37] RAY, N., VALLET, B., LI, W., LÉVY, B. "N-Symmetry Direction Field Design", *ACM Trans. Graph.*, v. 27, n. 2, pp. 1–13, 2008. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1356682.1356683.

[38] KIM, Y., LEE, C. H., VARSHNEY, A. "Vertex-Transformation Streams", *Graph. Models*, v. 68, n. 4, pp. 371–383, 2006. ISSN: 1524-0703, doi: http://dx.doi.org/10.1016/j.gmod.2006.03.005.

[39] KAZHDAN, M., FUNKHOUSER, T., RUSINKIEWICZ, S. "Shape Matching and Anisotropy", *ACM Trans. Graph.*, v. 23, n. 3, pp. 623–629, 2004. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1015706.1015770.

[40] KAZHDAN, M., FUNKHOUSER, T., RUSINKIEWICZ, S. "Symmetry Descriptors and 3D Shape Matching". In: *SGP '04: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 115–123, New York, NY, USA, 2004. ACM. ISBN: 3-905673-13-4, doi: http://doi.acm.org/10.1145/1057432.1057448.

[41] TAL, A., ZUCKERBERGER, E. "Mesh Retrieval by Components". In: *GRAPP*, pp. 142–149, 2006, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.1900.

[42] LIU, R., ZHANG, H., SHAMIR, A., COHEN OR, D. "A Part-Aware Surface Metric for Shape Analysis", *Computer Graphics Forum (Proceedings of Eurographics)*, v. 28, n. 2, pp. 397–406, 2009, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.151.655.

[43] BERNER, A., BOKELOH, M., WAND, M., SCHILLING, A., SEIDEL, H. "A Graph-Based Approach to Symmetry Detection". In: *Symposium on Volume and Point-Based Graphics*, pp. 1–8, Los Angeles, CA, 2008. Eurographics Association, doi: http://dx.doi.org/10.2312/VG/VG-PBG08/001-008.

[44] GATZKE, T., GRIMM, C., GARLAND, M., ZELINKA, S. "Curvature Maps For Local Shape Comparison", *International Conference on Shape Modeling and Applications*, pp. 244–253, June 2005, doi: http://dx.doi.org/10.1109/SMI.2005.13.

[45] KARNI, Z., GOTSMAN, C. "Spectral Compression of Mesh Geometry". In: *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 279–286, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN: 1-58113-208-5, doi: http://doi.acm.org/10.1145/344779.344924.

[46] GRINSPUN, E., SCHRÖDER, P., DESBRUN, M. *Discrete Differential Geometry: An Applied Introduction*. Boston, US, ACM SIGGRAPH 2006 Courses, 2006. Available at: <`http://ddg.cs.columbia.edu/`>.

[47] VALLET, B., LÉVY, B. "Spectral Geometry Processing with Manifold Harmonics", *Computer Graphics Forum (Proceedings of Eurographics)*,

v. 27, n. 2, pp. 251–260, 2008, doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01122.x.

[48] SIMARI, P., KALOGERAKIS, E., SINGH, K. "Folding meshes: Hierarchical mesh segmentation based on planar symmetry". In: *SGP '06: Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 111–119, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN: 30905673-36-3, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.5100.

[49] CHENG, Z., DANG, G., JIN, S. "A Meaningful Mesh Segmentation Based on Local Self-similarity Analysis", *10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pp. 288–293, Oct. 2007, doi: http://dx.doi.org/10.1109/CADCG.2007.4407896.

[50] MITRA, N. J., GUIBAS, L. J., PAULY, M. "Partial and approximate symmetry detection for 3D geometry", *ACM Trans. Graph.*, v. 25, n. 3, pp. 560–568, 2006. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1141911.1141924.

[51] YOSHIZAWA, S., BELYAEV, A., SEIDEL, H. "Smoothing by Example: Mesh Denoising by Averaging with Similarity-Based Weights". In: *International Conference on Shape Modeling and Applications*, p. 9, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 0-7695-2591-1, doi: http://dx.doi.org/10.1109/SMI.2006.38.

[52] SCHALL, O., BELYAEV, A., SEIDEL, H. "Feature-preserving non-local denoising of static and time-varying range data". In: *SPM '07: Proceedings of the ACM Symposium on Solid and Physical Modeling*, pp. 217–222, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-666-0, doi: http://doi.acm.org/10.1145/1236246.1236277.

[53] RIBEIRO, S., MAXIMO, A., BENTES, C., OLIVEIRA, A., FARIAS, R. "Memory-Aware and Efficient Ray-Casting Algorithm", *Computer Graphics and Image Processing, Brazilian Symposium on*, pp. 147–154, 2007. ISSN: 1530-1834, doi: http://doi.ieeecomputersociety.org/10.1109/SIBGRAPI.2007.28.

[54] PINA, A., BENTES, C., FARIAS, R. "Memory Efficient and Robust Software Implementation of the Raycast Algorithm". In: *WSCG'07: The 15th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2007.

[55] MAXIMO, A., RIBEIRO, S., BENTES, C., OLIVEIRA, A., FARIAS, R. "Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering". In: *VG-PBG*, pp. 155–162, Los Angeles, California, USA, 2008. Eurographics Association. ISBN: 978-3-905674-12-5, doi: http://doi.ieeecomputersociety.org/10.2312/VG/VG-PBG08/155-162.

[56] MAXIMO, A., MARROQUIM, R., FARIAS, R., ESPERANÇA, C. "GPU-Based Cell Projection for Large Structured Data Sets". In: *GRAPP (GM/R)*, pp. 312–322. INSTICC – Institute for Systems and Technologies of Information, Control and Communication, 2007. ISBN: 978-972-8865-71-9, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.6833.

[57] MAXIMO, A. *Projeção de Células baseada em GPU para Visualização Interativa de Volumes*. M.Sc. Dissertation, UFRJ – Federal University of Rio de Janeiro, RJ, Brazil, Nov 2006. Available at: <`http://www.lcg.ufrj.br/Members/andream`>.

[58] WILLIAMS, P. L. "Visibility-Ordering Meshed Polyhedra", *ACM Trans. Graph.*, v. 11, n. 2, pp. 103–126, 1992. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/130826.130899.

[59] MARROQUIM, R., MAXIMO, A., FARIAS, R., ESPERANÇA, C. "Volume and Isosurface Rendering with GPU-Accelerated Cell Projection", *Computer Graphics Forum*, v. 27, pp. 24–35, 2008. ISSN: 0167-7055, doi: http://dx.doi.org/10.1111/j.1467-8659.2007.01038.x.

[60] PHONG, B. T. "Illumination for Computer Generated Pictures", *Commun. ACM*, v. 18, n. 6, pp. 311–317, 1975. ISSN: 0001-0782, doi: http://doi.acm.org/10.1145/360825.360839.

[61] KLEIN, T., STEGMAIER, S., ERTL, T. "Hardware-Accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids". In: *PG '04: Proceedings of the 12th Pacific Conference on Computer Graphics and Applications*, pp. 186–195, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7695-2234-3, doi: http://doi.ieeecomputersociety.org/10.1109/PCCGA.2004.1348349.

[62] PASCUCCI, V. "Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping". In: *Joint Eurographics - IEEE VGTC Symposium on Visualization (VisSym)*, pp. 293–300, 2004, doi: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.6156.

[63] MAXIMO, A., MARROQUIM, R., FARIAS, R. "Hardware-Assisted Projected Tetrahedra", *Computer Graphics Forum*, v. 29, pp. 903–912, 2010. ISSN: 0167-7055, doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01673.x.

[64] CEDERMAN, D., TSIGAS, P. "A Practical Quicksort Algorithm for Graphics Processors". In: *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pp. 246–258, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN: 978-3-540-87743-1, doi: http://dx.doi.org/10.1007/978-3-540-87744-8_21.

[65] PLAUGER, P. J., LEE, M., MUSSER, D., STEPANOV, A. A. *C++ Standard Template Library*. Upper Saddle River, NJ, USA, Prentice Hall PTR, 2000. ISBN: 0134376331.

[66] "CUDA SDK Bitonic Sort Example". . nVidia™ CUDA, 2007. Available at: <`http://developer.download.nvidia.com/compute/cuda/sdk`>.

[67] TREECE, G. M., PRAGER, R. W., GEE, A. H. "Regularised Marching Tetrahedra: Improved Iso-Surface Extraction", *Computers & Graphics*, v. 23, n. 4, pp. 583–598, 1999, doi: http://dx.doi.org/10.1016/S0097-8493(99)00076-X.

[68] TAUBIN, G. "A Signal Processing Approach to Fair Surface Design". In: *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 351–358, New York, NY, USA, 1995. ACM. ISBN: 0-89791-701-4, doi: http://doi.acm.org/10.1145/218380.218473.

[69] LEE, C. H., VARSHNEY, A., JACOBS, D. W. "Mesh Saliency", *ACM Trans. Graph.*, v. 24, n. 3, pp. 659–666, 2005. ISSN: 0730-0301, doi: http://doi.acm.org/10.1145/1073204.1073244.

[70] ZERNIKE, F. "Beugungstheorie des Schneidenverfahrens und seiner verbesserten Form, der Phasenkontrastmethode", *Physica 1*, pp. 689–704, 1934.

[71] KHOTANZAD, A., HONG, Y. H. "Rotation invariant pattern recognition using Zernike moments". In: *9th International Conference on Pattern Recognition*, v. 1, pp. 326–328, Nov 1988, doi: http://dx.doi.org/10.1109/ICPR.1988.28233.

[72] REVAUD, J., LAVOUE, G., BASKURT, A. "Improving Zernike Moments Comparison for Optimal Similarity and Rotation Angle Re-

trieval", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 31, pp. 627–636, 2008. ISSN: 0162-8828, doi: http://doi.ieeecomputersociety.org/10.1109/TPAMI.2008.115.

[73] FLOATER, M. S. "Mean Value Coordinates", *Comput. Aided Geom. Des.*, v. 20, n. 1, pp. 19–27, 2003. ISSN: 0167-8396, doi: http://dx.doi.org/10.1016/S0167-8396(02)00002-5.

[74] "OpenGL Utility Toolkit". . GLUT, 2010. Available at: <`http://www.opengl.org/resources/libraries/glut`>.

[75] "CGAL, Computational Geometry Algorithms Library". . CGAL, 2010. Available at: <`http://www.cgal.org/`>.

[76] "Boost C++ Libraries". . Boost, 2010. Available at: <`http://www.boost.org/`>.

[77] "Visual Computing Lab Library". . VCGLib, 2010. Available at: <`http://vcg.sourceforge.net/`>.

[78] "Toolkit do Laboratório de Computação Gráfica". . LCGtk, 2010. Available at: <`http://code.google.com/p/lcgtk/`>.

[79] MAXIMO, A., PATRO, R., VARSHNEY, A., FARIAS, R. "A Robust and Rotationally Invariant Local Surface Descriptor with Applications to Non-local Mesh Processing", *Graphical Models (to appear)*, 2010. ISSN: 1524-0703.

[80] FARIAS, R., MITCHELL, J. S. B., SILVA, C. T. "ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering". In: *VVS'00: Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pp. 91–99, New York, NY, USA, 2000. ACM Press. ISBN: 1-58113-308-1, doi: http://doi.acm.org/10.1145/353888.353905.