REAL TIME RENDERING OF DEFORMABLE AND SEMI-TRANSPARENT OBJECTS BY VOLUME RENDERING

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Dipl.-Ing. Amel Guetat
aus Tunesien

Mannheim, 2008

Dekan:	Professor Dr. Matthias Krause, Universität Mannheim
Referent:	Professor Dr. Jürgen Hesser, Universität Heidelberg
Korreferent:	Professor Dr. Reinhard Männer, Universität Heidelberg

Tag der mündlichen Prüfung: 30. Juni 2008

Abstract

Volume rendering is one of the key technique to display data from diverse application fields like medicine, industrial quality control, and numerical simulations in an appropriate way. The current main limitations are still the inadequate rendering speed and the limited flexibility of the most efficient algorithms. In this dissertation, we developed three new algorithms for the acceleration of direct volume rendering and volume deformation. The first algorithm consists on a first step, on the reimplementation of the existing preintegration volume rendering approach, where the gray values between two sampling points change linearly, by considering the correct not simplified volume rendering integral, i.e, considering the attenuation factor as well as the shading function during the precompuation process. On a second step, we extended our algorithm to quadratic and higher order polynomial model. The preintegration speed for linear model is increased by a factor of 10. The second algorithm accelerates shear warp and ray casting process. While acceleration techniques like space leaping and early ray termination are efficient when rendering volumes with most of the voxels are mapped either opaque or transparent, encoding coherence appeared more efficient for rendering semi-transparent volumes. It's an approach for coding empty regions to a coherency encoding that can describe regions where the opacity changes linearly. We reimplemented this technique using a volume graphics library (VGL). We improved it by using the preintegration technique to evaluate opacity and shading inside the coherent region. We achieved a speedup of up to a factor of 3. The third algorithm is for volume deformation. The applied technique is the ray deformation where the volume deforming and the volume rendering are incorporated into a single process. This is implemented in our approach, by combining the Free Form Deformation (FFD) and inverse ray deformation. Unlike the previous implementation, our opacity and shading calculation are based on the preintegration technique which allows us to handle different lengths of the sampled intervals in the polyline segments which approximate the deformed ray.

Key Words: Volume rendering, Volume deformation, Pre-integration, Ray casting, Shear warp, Free Form Deformation, Coherence encoding.

Acknowledgements

First and foremost I should thank my supervisor Prof. Jürgen Hesser not only for all that I have learned from him during my PhD program at the Institute of Computational Medicine (ICM), but also for his full support in every aspect as a great supervisor. There is no need to mention that a big part of this thesis is the result of joint work with him.

I am deeply grateful to Prof. Reinhard Männer the head of the Chair of Computer Science V at the University of Mannheim and the director of the Institute of Computational Medicine (ICM), for providing me a very good working environment.

I would like to thank all my colleagues at the ICM for providing a good working atmosphere, and for the good time to relax from the work.

I wish to extend my warmest thanks to Christiane Glasbrenner, Andrea Seeger and Dina Goerlitz who have helped me with the administration effort, and to Christof Poliwoda from Volume Graphics for supplying me with his library which I have used during my work at the ICM.

I wish to thank my family: my sister Henda and my parents, Abdelmejid and Noura for their prayers and their encouragements throughout my graduate work in Germany. I thank them for their love, their support, and their confidence throughout the past twenty-eight years. I dedicate this work to them, to honor their love, patience, and support during these years.

Finally, I wish to express my loving thanks to my husband Skander who supported me in my work whenever he could.

My work at ICM was supported in part by a graduation scholarship after the national graduating promotion law (LGFG). I would like to express my gratitude for their support.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Volume visualization is very important for understanding the real world and getting information from it. It helps us to understand complex 3D data. That data can be obtained by different techniques. An example is the complex simulations for estimating wind resistance over the surface of a car. Another example is the geological research, in which acoustic waves generated by an artificial explosive are sampled and used to produce 3D maps of geological structures below the surface of the earth. Another typical example is the medical field, medical imaging technologies such as magnetic resonance (MRI) and computed tomography (CT) can produce a sequence of 2D slices which forms a 3D volume containing detailed representations of internal organs, allowing doctors to make a diagnosis without invasive surgery. The method which allows to reveal the information embedded in the tomography images is called volume rendering.

The current main limitations of the existing volume rendering algorithms are the inadequate rendering speed due to the huge size of the volume data.

1.1 Motivation

One main motivation of this work is to avoid artifacts in rendered images, due to volume data sets mapped with complex transfer functions. A pre-integration volume rendering technique [10] is a solution for such problem. It assumes a linear distribution of gray values between two consecutive slices, calculates the pre-integrated volume rendering integral for each possible combination in the transfer function and save the results in a lookup table. In our work, we give a comprehensive approach which describes the precomputation without simplifications and where the normals depend on the opacity instead of the gray values. We achieved a speedup of up to a factor of 3. We also extended this technique to quadratic, and higher-order polynomial model for the change of the gray values between two sample points. The second motivation of this work is to support applications where one deals with data sets that are acquired from objects with complicated inner structures. In order to see both, inner structures and their relation to the outer shape of the object, one typically uses semi-transparency. However, with this parameter setting, one has to take into consideration that the overall performance of the rendering system drops dramatically. In order to overcome this problem, a coherency encoding approach [5] was suggested to describe regions where the opacity change linearly. This can give a considerable speedup of up to a factor of 2. A first prototype showed that this approach is feasible in principle but provides no real-time encoding and visualization. In order to solve this problem, we proposed to combine the coherence encoding approach with the pre-integrated volume rendering to evaluate opacity and shading inside the coherent region. We achieved a speedup of up to a factor of 3. The third main motivation is that we are able to render deformed volumes that occur in all cases where one performs numerical simulations e.g. in computational fluid dynamics, structural mechanics etc. For example medical training system where the physician can perform a virtual operation that is such real-looking that during the intervention one cannot distinguish between reality and simulation. The applied technique is the ray-deformation concept. This concept has, in principle, demonstrated to work but there are still many open problems for their real usage. The first is the significant overhead to transform the local image gradient according to the deformation field. We overcome this problem by considering a polynomial approximation of the deformed ray. We also remedied to the problem of opacity compensation for volume change by evaluating the opacities directly from the opacity lookup tables calculated for different distances, in the pre-integrated preprocessing step.

1.2 Thesis Outline

The work presented in this thesis is distributed over five main parts:
Chapters 2-4 belongs to the first part, where chapters 2 and 3 present the results of a thorough investigation into the current trends in volume graphics and volume rendering. Chapter 4 gives an overview about the software used for our development namely the Volume Graphics Library (VGL).
The second part consists on the chapter 5, where we developed the improved pre-integration technique. We first implemented it considering no simplification for volume rendering integral and for different sampling interval distances. We further extended this technique to quadratic and higher order polynomial model for gray value change inside the volume data.
The third part which is the chapter 6 is about encoding coherence in volume data.

In a first step, we implement the existing technique on the shear warp algorithm, considering two voxel scanlines simultaneously when marching through the volume. In a second step, we apply the coherence encoding technique to the ray casting algorithm. For that purpose, we first evaluated for each voxel in the volume its coherence distance, using a Taylor-expansion based spatial coherence method and the result is saved in a 3D array data structure. In a second step, we address those spatial coherence information to encode coherency along the ray traversal. And we directly retrieve opacity and shading values in the coherent regions from the pre-integrated lookup tables with the corresponding distances.
The fourth part is the chapter 7. It addresses the volume deformation algorithm. We first gave an overview about the theoretical background of the deformation method. Further, we developed the deformation method which combines inverse ray deformation with the uniform B-spline representation of the free form deformations. We optimize the existing method by approximating the deformed ray by a piecewise second degree polynomial calculated explicitly thanks to the quadratic uniform B-spline. This will avoid us recalculating at each time the FFD transform which is time consuming task. Another improvement is for opacity compensation. After the deformation, the density within the volume is changed, therefore the opacity value of the sample points should be corrected to reflect the change of volume density. We addressed this problem by reading directly the correct opacity value in the pre-integrated opacity lookup table calculated in a preprocessing step and corresponding to the new sampling interval distance of the deformed ray.

To close, a brief summary of the undertaken work is given in the last part represented by the chapter 8.

Chapter 2

Background of Volume Graphics

2.1 Introduction

For the analysis and interpretation of eventual information contained within a volume data set, the exploration of the data is required. This is achieved by volume visualization techniques. Volume visualization is a process of transforming information into a visual form enabling the user to observe the information. It is concerned with volume data representation, modeling, manipulation and rendering. In this chapter, we begin with an introduction to volume data and then a description of the volume visualization process.

2.2 Volumetric Data

2.2.1 Volume Data Acquisition

Volume data can be acquired either by sampling, simulation or modeling techniques. For example, a sequence of 2D slices obtained from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) is 3D reconstructed into a volume model for medical examination or surgery. CT is also used for non-invasive inspection of mechanical parts. Similarly, confocal microscopes produce data sets which are visualized to study the morphology of biological structures. In computational fluid dynamics, the result of simulation are often visualized as volume data for analysis and verification.

2.2.2 Volume Data Representation

Mathematically speaking, volumetric data is typically a set S of samples (x, y, z, v), representing the value v of some property of the underlying object at the 3D location (x, y, z). The value v can represent one or a group of measurable properties like density, color, heat or pressure. A volume is exactly such a 3D model and it stores the contents of a 3D object by a 3D lattice of points. The element of the 3D lattice is called voxel and stores v. The voxels are typically defined on a regular grid, a 3D array in computer memory. For this reason, S will be referred to as the array of values $S(x, y, z)$. Alternatively, either rectilinear, curvilinear(structured), or unstructured grids are employed(Figure 2.1). An unstructured or irregular volume is a collection of cells whose connectivity has to be specified explicitly. These cells can be of arbitrary shape such as tetrahedron, hexahedron or prism.

2.2.3 Volume Resampling

The array S only defines the value of some measured property of the data at discrete locations in space. To determine values at arbitrary sample points location, volume resampling is needed. It consists on applying some interpolation function to S. There are many possible interpolation functions. The simplest one is the nearest neighbor

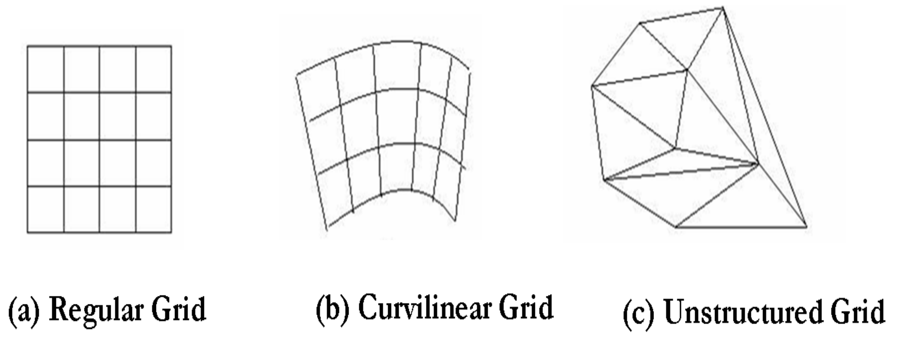


Figure 2.1: Example Grid Types

interpolation, it has the order zero: the value at any location in $\mathbb{R}^3$ is simply the value of the closest sample to that location. This interpolation method is known for its low computational complexity but on the other hand, a low quality image results. A first order interpolation method can also be used, known as trilinear interpolation. With this interpolation function, the value v is assumed to vary linearly along directions parallel to one of the major axes. Let the point P lie at location (x_p, y_p, z_p) within a voxel defined by samples $V_{000}, V_{100}, V_{010}, ...etc....V_{111}$. For simplicity, let the distance between samples in all three directions be 1. The value at position P within the cube will be denoted v_p, as shown in figure 2.2 and is given by:

$v_p = V_{000}(1-x_p)(1-y_p)(1-z_p) + V_{100}x_p(1-y_p)(1-z_p) + V_{010}(1-x_p)y_p(1-z_p) + V_{001}(1-x_p)(1-y_p)z_p + V_{101}x_p(1-y_p)z_p + V_{011}(1-x_p)y_pz_p + V_{110}x_py_p(1-z_p) + V_{111}x_py_pz_p$.

Higher-order interpolation functions can also be used like cubic spline interpolation. The cubic spline interpolation is a piecewise continuous curve, passing through each of the values in the array. There is a separate cubic polynomial for each interval, each with its own coefficients. The value of a sample point $x \in [x_i, x_{i+1}]$ is given by:

$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$.

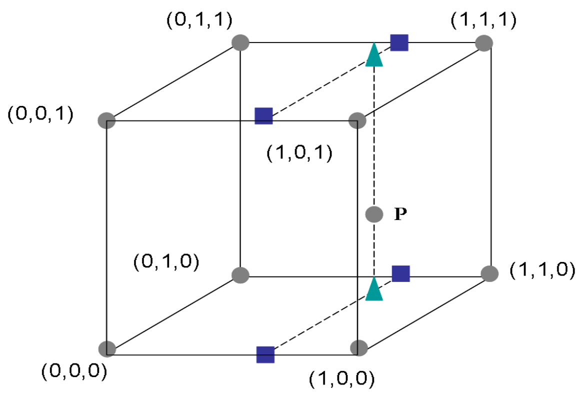


Figure 2.2: Tri-Linear Interpolation

Together, these polynomial segments are denoted $S(x)$: the spline. Four coefficients for each interval are required to define the spline $S(x)$.

2.2.4 Volume Visualization

Over the years, many techniques have been developed to visualize 3D data. Volume visualization is used to create images from scalar and vector data sets defined on multiple dimensional grids, i.e., it is the process of projecting a multidimensional (usually 3D) data set onto a 2D image plane to gain an understanding of the structure contained within the data.

The typical volume visualization process can be described with figure 2.3. First,

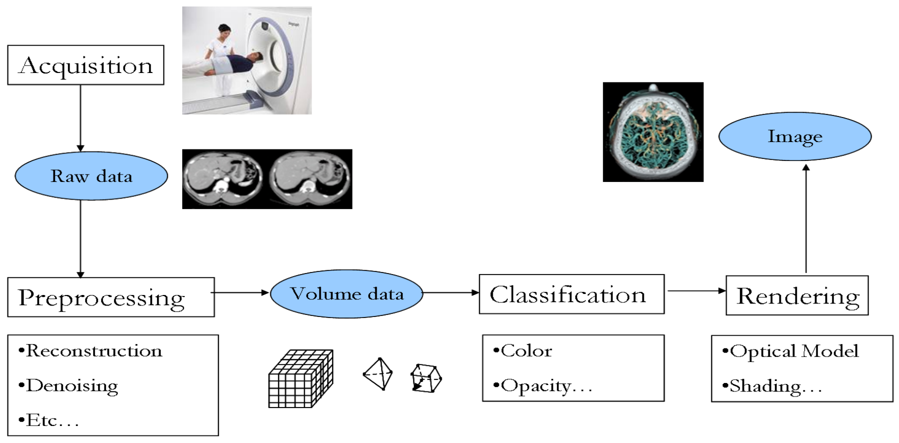


Figure 2.3: Visualization Process

data acquisition is performed either via empirical measurement or computer simulation to obtain a raw data which have to be preprocessed for use. Data preprocessing describes any type of processing on raw data which is transformed into a format(volume data) that will be more easily and effectively processed for the purpose of the user. There are different methods used for preprocessing, including sampling, which selects a representative subset from a large population of data, scaling the data for a better value distribution and denoising, which removes noise from data. Therefore, the volume data is classified. Classification is the process of mapping the scalar values in the volume to primary colors and opacities. This is usually defined using an RGBA transfer functions. The settings of the color and opacity depends on the information the user want to reveal. After classification, the volume is rendered and finally displayed. In the next section, we describe in details the volume rendering process.

2.3 Volume Rendering

Volume rendering is a technique for displaying volumetric data sets, especially 3D scalar data fields, to model and understand the natural scenes containing clouds, fog, flames and so on. In order to get most realistic results, it is important to model these features correctly. In this thesis, we consider the reconstruction of images of some scenes as it would be recorded by a photographic camera. This can be completely described by the known physical laws of optics. In order to simulate this process, one has to calculate the radiation field as it would be produced by a real scene. Thereby, things will be much easier if one ignores the wave character of light and its two possible states of polarization. In fact, this approximation is most often used in praxis. Neglecting these effects we are dealing with geometrical optics in contrast to physical optics of light. Considering the approximation of geometrical optics, the interaction of light with surfaces and volume elements can be completely described within the framework of linear transport theory [4, 3]. In the first part of this section, we will discuss the basics of this theory. In the second part, we will subsequently discuss various techniques for solving the central equation of transport theory, the equation of transfer. Different solution techniques directly translate into different rendering algorithms which will be discussed later, in the next chapter.

2.3.1 Transport Theory of light

The interaction of light with surfaces and volume elements [30, 50, 32] is properly described by methods of *radiation transport theory*. In the following, we discuss the basis of radiation transport theory and the derivation of the equation of transfer.

2.3.1.1 Radiant Energy

Radiometry deals with the description of light on a transport theoretic level. We note that light can also be measured using the techniques of photometry which deals with brightness as perceived by the human eye rather than the absolute power. In computer graphics, we are interested in the measurement of the intensity of the light instead of the number of photons. Using intensity, $I(\vec{x}, \vec{n}, \nu)$ completely describes the radiation field at any point $\vec{x}$, giving both its distribution in angle and frequency. Of course, the radiation field has also a time dependance but here, we will assume the radiation field to be time independent, because the photon flow in a limited space is assumed to reach equilibrium almost instantaneously due to the large velocity of light. The amount of radiant energy δE, traveling in time dt within a frequency interval $d\nu$ around ν through a surface element da into an element of solid angle $d\Omega$ in direction $\vec{n}$, is given by:

$$\delta E = I(\vec{x}, \vec{n}, \nu) cos\vartheta da d\Omega d\nu dt \tag{2.3.1}$$

Here ϑ is the angle between $\vec{n}$ and the normal on da. More formally, the intensity I is also called *radiance*. It is the power density transmitted by the photons at the position $\vec{x}$ in direction $\vec{n}$, and it is measured in units of Watt per meter squared per solid angle per frequency ($Wm^{-2}sr^{-1}$).

2.3.1.2 Absorption, Emission, and Scattering

When radiation passes through material, it undergoes several changes. First, an amount of energy is generally removed from a beam of intensity $I(\vec{x}, \vec{n}, \nu)$. We describe this loss due to the absorption in terms of absorption coefficient $\chi(\vec{x}, \vec{n}, \nu)$. It is composed of a *true absorption coefficient* $\kappa(\vec{x}, \vec{n}, \nu)$ and a *scattering coefficient* $\sigma(\vec{x}, \vec{n}, \nu)$:

$$\chi = \kappa + \sigma. \tag{2.3.2}$$

This energy when passing through a volume element of length ds and cross area da, is given by:

$$\delta E_{ab} = \chi(\vec{x}, \vec{n}, \nu) I(\vec{x}, \vec{n}, \nu) ds da d\Omega d\nu dt. \tag{2.3.3}$$

Second, energy can be emitted and this can be described by the emission coefficient $\eta(\vec{x}, \vec{n}, \nu)$. This coefficient is composed of a *thermal part or source term* $q(\vec{x}, \vec{n}, \nu)$ and a *scattering part* $j(\vec{x}, \vec{n}, \nu)$:

$$\eta = q + j. \tag{2.3.4}$$

The total emission coefficient is defined in such a way, that the amount of radiant energy within a frequency interval da emitted in time dt by a volume element of length ds and cross section da into a solid angle dn in a direction $\vec{n}$ is:

$$\delta E_{em} = \eta(\vec{x}, \vec{n}, \nu) ds da d\Omega d\nu dt. \tag{2.3.5}$$

2.3.1.3 Equation of transfer

The change of the intensity due to absorption and emission, is described by the equation of transfer. Considering the volume element in figure 2.4, the difference

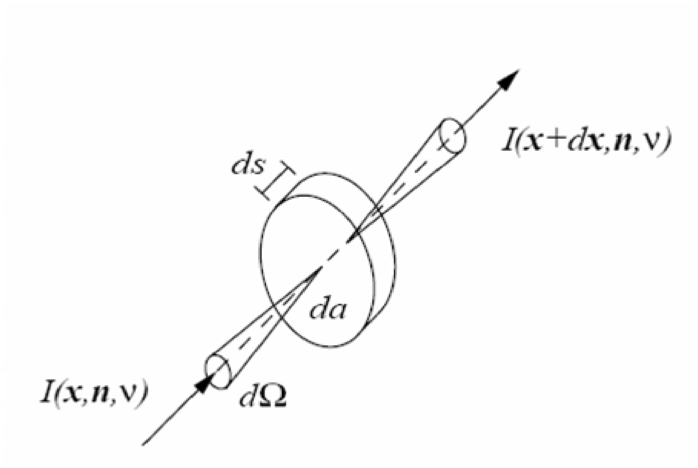


Figure 2.4: Light-material interaction

between the amount of energy emerging at position $\vec{x} + d\vec{x}$ and the amount of energy incident at $\vec{x}$ must be equal to:

$$\begin{aligned} I(\vec{x}, \vec{n}, \nu) - I(\vec{x} + d\vec{x}, \vec{n}, \nu) dad\Omega d\nu dt &= \\ -\chi(\vec{x}, \vec{n}, \nu) I(\vec{x}, \vec{n}, \nu) + \eta(\vec{x}, \vec{n}, \nu) dsdad\Omega d\nu dt \end{aligned} \tag{2.3.6}$$

By writing $dx = nds$, we immediately obtain the time independent equation of transfer:

$$\vec{n} \cdot \nabla I = -\chi I + \eta, \tag{2.3.7}$$

where we have used the directional derivative:

$$\vec{n} \cdot \nabla I = n_x \frac{\partial I}{\partial x} + n_y \frac{\partial I}{\partial y} + n_z \frac{\partial I}{\partial z} = lim_{\Delta s \to 0} \frac{I(\vec{x}) - I(\vec{x} + \vec{n}\Delta s)}{\Delta s} \tag{2.3.8}$$

2.3.1.4 Boundary Conditions

Like for other differential equations, we have to specify some boundary conditions, to completely describe the radiation field. Let us describe what happens at the surfaces $\mathcal{S}$. For opaque surfaces, they are assumed to be a surface emitter E. We have then the following *explicit boundary condition*:

$$I(\vec{x},\vec{n},\nu) = E(\vec{x},\vec{n},\nu), \vec{x} \in \mathcal{S} \tag{2.3.9}$$

where $\vec{n}$ always points into the volume. Explicit boundaries are independent of I itself. In contrast, implicit or reflecting boundary conditions are defined by:

$$I(\vec{x},\vec{n},\nu) = \int\int k(\vec{x},\vec{n}',\vec{n},\nu',\nu) I(\vec{x},\vec{n}',\nu') d\Omega' d\nu', \vec{x} \in \mathcal{S} \tag{2.3.10}$$

where k is the surface scattering kernel.

Thus, these boundary conditions leads to the well-known rendering equation, as pointed out by Kajiya [30].

2.3.1.5 Integral form of the equation of transfer

Let us now integrate equation 2.3.7 along a ray. First, we notice that the operator $\vec{n} \cdot \nabla$ is the directional derivative along a line $\vec{x} = \vec{p} + s \cdot \vec{n}$, with $\vec{p}$ being some arbitrary reference point. Thus, the equation of transfer 2.3.8 can be rewritten as:

$$\frac{\partial}{\partial s} I(\vec{x},\vec{n},\nu) = -\chi(\vec{x},\vec{n},\nu) I(\vec{x},\vec{n},\nu) + \eta(\vec{x},\vec{n},\nu), \tag{2.3.11}$$

The *optical depth* between two points $\vec{x_1} = \vec{p} + s_1 \cdot \vec{n}$ and $\vec{x_2} = \vec{p} + s_2 \cdot \vec{n}$ is defined as:

$$\tau_\nu(\vec{x_1}, \vec{x_2}) = \int_{s_1}^{s_2} \chi(\vec{p} + s' \cdot \vec{n}, \vec{n}, \nu) ds' \tag{2.3.12}$$

Notice that equation 2.3.11 has an integrating factor $e^{\tau_\nu(x_0,x)}$ and thus can be written as:

$$\frac{\partial}{\partial s}(I(\vec{x}, \vec{n}, \nu)e^{\tau_\nu(\vec{x_0},\vec{x})}) = \eta(\vec{x}, \vec{n}, \nu)e^{\tau_\nu(\vec{x_0},\vec{x})} \tag{2.3.13}$$

By integrating, we get:

$$I(\vec{x}, \vec{n}, \nu)e^{\tau_\nu(\vec{x_0},\vec{x})} - I(\vec{x_0}, \vec{n}, \nu) = \int_{s_0}^{s} \eta(\vec{x'}, \vec{n}, \nu)e^{\tau_\nu(\vec{x_0},\vec{x'})} ds' \tag{2.3.14}$$

This equation can be considered as the general formal solution of the equation of transfer. It states that the intensity of radiation traveling along $\vec{n}$ at point $\vec{x}$ is the sum of photons emitted from all points along the line segment $\vec{x'} = \vec{p} + s' \cdot \vec{n}$, attenuated by the integrated absorption of the intervening material, plus an attenuated contribution from photons entering the boundary surface when it is pierced by that line segment.

2.3.2 Volume Rendering Equation

From the above equation 2.3.14, we can obtain the famous volume rendering equation, which is the basis for all volume rendering. For this purpose, different assumptions are stated to solve this equation.

First, according to the *emission-absorption model*[39, 34, 26], scattering is ignored. Then the emission coefficient is equivalent to: $\eta = q$. Similarly, the so-called low-albedo model is used to only consider the κ coefficient in the absorption term i.e $\chi = \kappa$. Second, because no mixing between different frequencies is possible, we can assume that the emission and the intensity do not depend on the frequency ν. Finally, we choose a different set of variables by replacing $\vec{x}$ and $\vec{n}$ by s. The equation of transfer in integral form is then:

$$I(s) = I(s_0)e^{-\tau(s_0,s)} + \int_{s_0}^{s} q(s')e^{-\tau(s',s)}ds' \tag{2.3.15}$$

with optical depth

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(s)ds \tag{2.3.16}$$

Equation 2.3.15 is the volume rendering equation: The intensity measured at position s is composed of the intensity of the background $I(s_0)$, given by the boundary conditions, that is reduced by the absorption between s_0 and s. The second component is reflection, i.e. at each point s', the amount $q(s')$ is reflected to the viewer direction and this light is absorbed from s' to s.

The volume rendering equation is solved numerically by discretizing along the ray path with sample points s_k. We divide the range of integration into n intervals as shown in figure 2.5, then the intensity at position s_k is obtained from the intensity s_{k-1} by substituting s_0 in equation 2.3.15 by s_{k-1}:

$$I(s_k) = I(s_{k-1})e^{-\tau(s_{k-1},s_k)} + \int_{s_{k-1}}^{s_k} q(s)e^{-\tau(s,s_k)}ds \tag{2.3.17}$$

Let we introduce the following substitutions:

$$\theta_k = e^{-\tau(s_{k-1},s_k)} \tag{2.3.18}$$

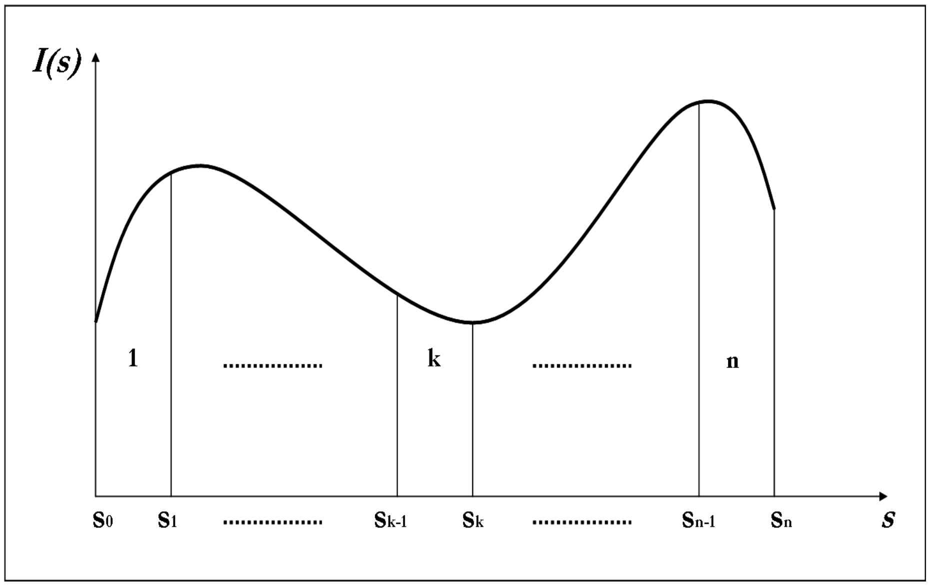


Figure 2.5: Discretizing the ray trajectory.

and

$$b_k = \int_{s_{k-1}}^{s_k} q(s) e^{-\tau(s,s_k)} ds \tag{2.3.19}$$

θ_k and b_k are respectively the transparency and the color of the material in between the interval $[s_{k-1}, s_k]$. Transparency is defined between 0 and 1. Usually we can use the opacity denoted by α and defined by: $\alpha_k = 1 - \theta_k$.

Thus, the intensity at a sample point s_n can be written as:

$$\begin{aligned} I(s_n) &= I(s_{n-1})\theta_n + b_n = (I(s_{n-2})\theta_{n-1} + b_{n-1})\theta_n + b_n = ... \\ &= \sum_{k=0}^{n} b_k \prod_{j=k+1}^{n} \theta_j, \end{aligned} \tag{2.3.20}$$

We also can rewrite this formula by:

$$I(s_n) = \sum_{k=0}^{n} b_k \prod_{j=k+1}^{n} 1 - \alpha_j, \tag{2.3.21}$$

We thus end up with a formula that describes in a recursion equation how to generate an image, given the physical values on the sample points of the rays. This formula will be used in the next chapter to derive the implementations of volume rendering. To evaluate the ray intensity using the equation 2.3.21, the usual way is to use the rectangle rule. In case where the absorption and emission coefficients are explicitly described by a polynomial of certain degree, we can exploit this by using a higher order quadratic polynomial [26].

In volume rendering, the scalar values associated to each voxel are mapped to the parameters in the physical model like the absorption coefficient and the emission function. Opacity is obtained by mapping the scalar values of the voxels via a user defined opacity transfer function, while the contribution part of the emission namely the color is usually determined using a shading function. In the next section, we describe how simulate the illumination of light in volume.

2.3.3 Shading with the Phong Model

Illumination models simulate the interaction of the light with the objects. We distinguish two types of illumination model. On one hand, the local illumination model characterizes the contribution from the light that starts from a light source and is reflected on a surface. This means, the shading of any surface is independent from the shading of other surfaces. On the other hand, the global illumination model which adds to the previous one the light reflected from other surfaces to the current surface. This is by consequence more physically correct model and produces more realistic images but it is also more computationally expensive. In volume rendering, the local illumination model is preferred since the purpose is to display the shape of objects in volume data set rather than a correct phot-realism [22]. In fact, that was the assumption during generating the volume rendering equation. Phong reflection model is an illumination and shading model developed by Bui Tuong Phong [44]. It can produce a certain degree of realism by combining diffuse, specular and ambient light at a given point. We discuss these three types of reflection on the following.

2.3.3.1 Ambient Reflection

Ambient reflection is a gross approximation of multiple reflections from indirect light sources (e.g. the surfaces of walls and tables in a room that reflect off the lights from light sources). Ambient reflection produces a constant illumination on all surface, regardless of their orientation. The intensity of ambient reflection is proportional to the local properties of the object,which is modeled by a constant $K_a \in [0, 1]$, called ambient reflection coefficient. If we denote the intensity of the ambient light by I_a, the intensity of the ambient reflection is then given by:

$$I_{amb} = K_a I_a \tag{2.3.22}$$

2.3.3.2 Diffuse Reflection

Diffuse reflection is the uniform reflection of light with no directional dependence for the viewer. It originates from light coming directly from a source to a surface and then reflected to the viewer. The diffuse reflection is governed by the Lamberts law [60]. The intensity of the reflected light is proportional to the dot product of the surface normal and the light source direction, simulating a perfect diffuser and yielding a reasonable looking approximation to a dull, matte surface. The diffuse reflection is then:

$$I_{diff} = K_d I_i(\vec{N} \cdot \vec{L}) \tag{2.3.23}$$

where K_d is the diffuse reflection coefficient, I_i is the light reaching the local surface element. We assume that light source is infinitely distant so that I_i is constant for every voxel. $\vec{L}$ is the unit direction vector from the position of the considered point on the surface to the point light source. $\vec{N}$ is the unit normal vector of a surface. It is parallel to the local gradient of the voxel scalar value $g(x, y, z)$:

$$\vec{N}(x, y, z) = \frac{\nabla g(x, y, z)}{|\nabla g(x, y, z)|} \tag{2.3.24}$$

Often, the gradient is approximated using the central difference gradient operator:

$$\begin{aligned} \nabla g(x, y, z) &= \frac{1}{2}[g(x+1, y, z) - g(x-1, y, z)]\vec{i} \\ &+ \frac{1}{2}[g(x, y+1, z) - g(x, y-1, z)]\vec{j} \\ &+ \frac{1}{2}[g(x, y, z+1) - g(x, y, z-1)]\vec{k} \end{aligned} \tag{2.3.25}$$

Since gradient estimation is computational expensive, instead of estimating the gradient on the fly, usually it is precomputed for each voxel and saved with it. This method necessitates a large memory space. To overcome this problem, Lacroute [34] proposed to precompute gradient vectors and encode them into a look up table which is indexed with a convenient integer.

2.3.3.3 Specular Reflection

Specular reflection is when the reflection is stronger in one viewing direction, i.e. there is a bright spot, called a specular highlight. This is readily apparent on shiny surfaces. For an ideal reflector, such as a mirror, the angle of incidence equals the angle of specular reflection. Therefore, if $\vec{R}$ is the direction of specular reflection and $\vec{V}$ is the direction of the viewer, then for an ideal reflector the specular reflection is visible only when $\vec{V}$ and $\vec{R}$ coincide. For real objects (not perfect reflectors), the specular reflectance can be seen even if $\vec{V}$ and $\vec{R}$ do not coincide, i.e. it is visible over a range of a values α (or a cone of values). The shinier the surface, the smaller the α range for specular visibility. So a specular reflectance model must have maximum intensity at $\vec{R}$, with an intensity which decreases as α increases. In the Phong illumination model, this is modeled by a power n of the scalar product between $\vec{V}$ and $\vec{R}$. The specular reflection is then written as:

$$I_{spec} = K_s I_i (\vec{V}.\vec{R})^n \tag{2.3.26}$$

where K_s is the specular coefficient and n is the shininess exponent of the material. Hence, the whole Phong illumination formula is:

$$I = I_{amb} + I_{diff} + I_{spec}, \tag{2.3.27}$$

When we have color representation as RGB values, the equation 2.3.27 is calculated individually for R,G and B wave lengths.

2.4 Conclusion

In this chapter, we discussed the basis of volume rendering algorithms. We first presented the volume visualization process and then, we derived the volume rendering equation from the transport theory of light. We also discussed important aspects in volume rendering namely the shading operation.

In the next chapter, we will discuss in detail some algorithms for volume rendering and techniques to optimize them.

Chapter 3

Volume Rendering Algorithms

3.1 Introduction

In this chapter, we first describe the volume rendering process. Then, we present two important algorithms for volume rendering namely the ray-casting algorithm and the shear warp algorithms. Finally, we discuss techniques for acceleration and optimization of volume rendering algorithms.

3.2 Volume Rendering Process

As illustrated in figure 3.1, the volume rendering process consists of three main rendering phases [57]. Starting from a projection plane subdivided into screen pixels, a viewing ray is casted into the virtual scene, through each pixel. Each ray is then sampled at equidistant points, called sample points. The gray value of each sample point is interpolated using the gray values of the voxels in its neighborhood in the data set. After interpolation, gradients are estimated with the central difference gradient operator. In a second phase, using the Phong shading calculation, each sample point is assigned an intensity reflected into the direction of the observer. Finally, the

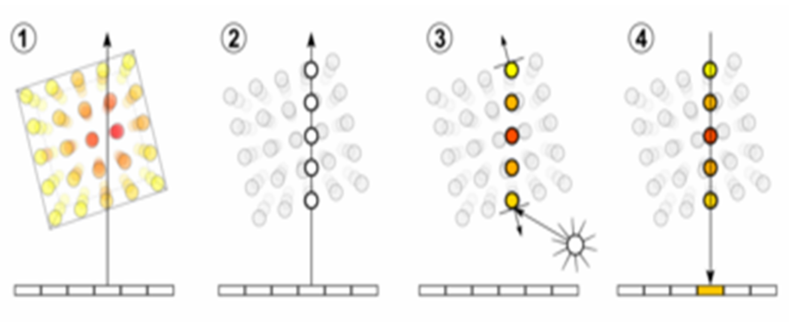


Figure 3.1: The four basic steps of volume rendering: (1)Ray-casting (2)Sampling (3)Shading (4)Compositing

third phase consists on compositing, for each ray, all contributions, by using the **over** operator. We discuss in more detail this phase in the following section.

3.2.1 Compositing Formula

Let us first recall the volume rendering recursive formula:

$$I(s_k) = I(s_{k-1})(1 - \alpha_k) + b_k \tag{3.2.1}$$

And noting $I(s_{k-1})$ by $\mathcal{C}_{in}$, $I(s_k)$ by $\mathcal{C}_{out}$, and the light that is reflected from a voxel by $b_k = \alpha_k \mathcal{C}_k$, the equation 3.2.1 can be rewritten as:

$$\mathcal{C}_{out} = \mathcal{C}_k \alpha_k + (1 - \alpha_k)\mathcal{C}_{in} \tag{3.2.2}$$

To abbreviate this formula, the *over* operator is introduced, we obtain:

$$\mathcal{C}_{out} = \mathcal{C}_k over \mathcal{C}_{in} \tag{3.2.3}$$

3.3 Volume Rendering Algorithms

Different direct volume rendering techniques (DVR) are introduced in the literature. In this chapter, we focus on the ray-casting approach as well as the shear warp algorithm.

3.3.1 Ray-Casting

Ray-casting is a famous volume rendering approach to produce high image quality [36]. It is an image based volume rendering technique, since the traversed order of the related voxels is determined by the pixels on the projection plan. The volume data set in ray-casting algorithm is viewed from a viewpoint through a view plane. From the view point, the ray-casting algorithm shots a ray through each pixel of the view plane into the volume. If the casted ray does not intersect the volume, it is simply skipped and the corresponding pixel is assigned with a default color usually black. Otherwise, the ray is sampled using trilinear interpolation from the voxels values of the volume. Each sample is then classified using transfer functions defined by the user. Finally, the samples are shaded and composited to the image plane as shown in figure 3.1.

The main disadvantage of the ray-casting is its computational cost due to the rendering process namely the shading calculation and the trilinear interpolation. Nevertheless, it exists different methods to accelerate ray-casting like early ray termination, space leaping and coherence encoding. These methods will be discussed in more details later in this chapter.

3.3.2 Shear Warp

The shear warp algorithm is one of the fastest algorithms for volume rendering. It was developed by Cameron and Undrill [2] and further popularized by Lacroute and Levoy [34]. In a preprocessing step, after the classification step, the shear warp algorithm computes, for each principle view direction, the run-length-encodings of the voxel data based on the opacity values. After that and before rendering step, the viewing matrix V is factorized into two matrixes: a shear matrix S and a warp one W and such as $V = W \cdot S$. Then, the principle viewing axis is determined from V and the run-length-encoding is computed. Further, the volume is sheared parallel to the set of slices that is most perpendicular to the viewing direction and then the sheared slices are resampled and composited into an intermediate image in a front to back order. During rendering, two adjacent runs are traversed simultaneously and the volume can be resampled using 2D interpolation. Space leaping is achieved efficiently thanks to the run-length array of the voxel scanlines. Each time, a non-transparent voxel is encountered in one of the two runs, two cases are possible. If the corresponding pixel in the intermediate image is not already opaque, it is updated. Otherwise, early ray termination is applied by skipping all the adjacent opaque pixels and the dynamic run-length-encoded data structure of the intermediate image is updated. Finally, after rendering all the slices, the intermediate image is warped into the final image using a 2D affine transformation (see figure 3.2).

The shear-warp algorithm is relatively fast in software at the cost of less accurate sampling which results in lower image quality compared to ray-casting. Acceleration methods for ray-casting such as early ray termination, space leaping, coherence encoding and pre-integrated volume rendering technique [56] can also be applied to

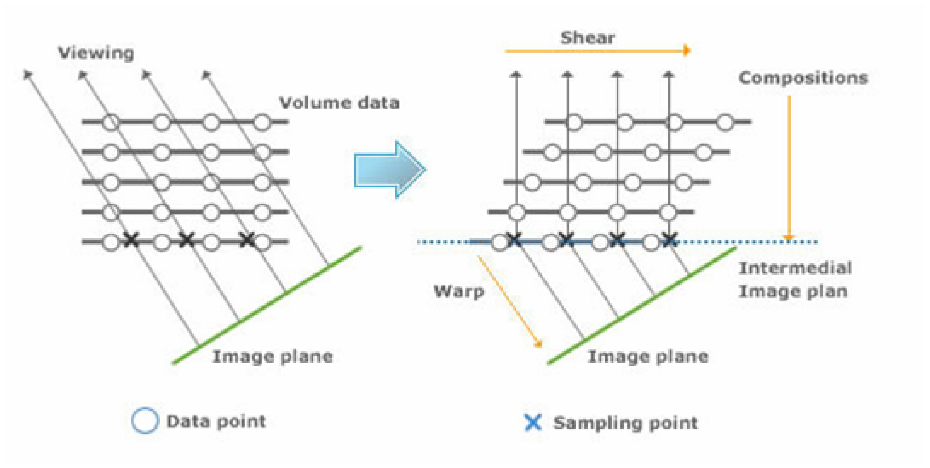


Figure 3.2: Shear-Warp transformation for parallel projection

shear-warp algorithm.

3.4 Volume Rendering Acceleration Methods

To achieve higher speed in volume rendering algorithms, many acceleration techniques have been proposed which can be achieved by either algorithmic improvement such as coherence acceleration using spatial data structures and exploiting homogeneity inside the volume, or parallelization. In the following, we begin with the standard techniques and then discuss further improvements.

3.4.1 Early Ray Termination

Early ray termination (ERT) is an acceleration technique for volume rendering. For the ray-casting algorithm, the ray is traced in front-to-back order and is terminated as soon as the accumulated ray opacity reaches a threshold close to full opacity. Then the rest of the voxels which would have been reached by the ray should be occluded and need not be rendered. The goal of this optimization technique is to eliminate samples in occluded regions of the volume, and thus saving the number of opaque voxels. Levoy [36] reports that a ray caster rendering medical data sets with early ray termination using a threshold of 95% could achieve speedups of $1.6 - 2.2\times$.

3.4.2 Space leaping

Space Leaping [63] is an effective branch of acceleration techniques for volume rendering. It is a method which provides an efficient traversal of the volume by skipping the empty spaces. A wide range of approaches for space leaping have been proposed such as:

- spatial subdivision: a hierarchical representation(e.g. octree, pyramid) of the volume which split this later into uniform regions that can be represented by nodes in a hierarchical data structures to skip the empty space when ray traversal[36]. A problem with this approach is the inefficient traversal of the data compared to regular volume traversal.
- Parallelization [29],[19].
- Distance coding [64].

In the following, we will not give an exhaustive overview of all techniques. We rather focus on the class of space leaping techniques which are related to our work namely space leaping which relies on a distance coding. Before rendering the volume, for each voxel, the distance to the next voxel having a gray value exceeding a given threshold (opaque), in its 3D neighborhood is determined and is stored in an extra distance

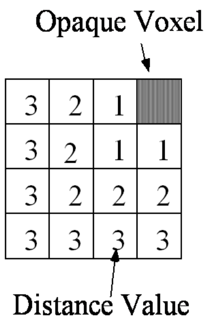


Figure 3.3: Each voxel contains a number that identifies its distance to the next opaque voxel.

data set with the same size as the volume. Those distances are view independent (see figure 3.3).

This preprocessing technique called distance coding is well suitable for ray-casting. It has been adapted, in another way, by Lacroute [34] to accelerate the shear-warp algorithm. In the shear-warp algorithm, the volume is encoded with run-lengths code. This structure is used when traversing the volume in a storage order, to skip the empty voxels in the volume scanlines i.e the opaque voxels are not composited. More details about this approach will be discussed in chapter 6.

3.4.3 Coherence Encoding

Further optimizations use not only transparent but also constant opacities [14] or extend this approach to linearly changing opacities [5]. The idea is to encode in a modified distance data set a measure of local linearity of the opacity function. And during rendering process, instead of saving all voxels which lie on a linear region, on term of opacity value, only the voxels at the boundary are saved and the opacity values of the voxels in between are obtained by linear interpolation. This approach was adapted for both ray-casting and shear-warp algorithm. We later focus on this acceleration method and improve the existing algorithm.

3.5 Conclusion

In this chapter, we presented two relevant volume rendering algorithms namely the ray-casting which achieves high quality images, and the shear-warp algorithm known for its higher speed performance. We then made a survey of some acceleration techniques such as early ray termination which terminates the ray when the accumulated opacity reaches a certain predefined threshold, space leaping which skips over the empty voxels on the volume, and coherence encoding which allows to encode the linearity inside a volume. In the next chapter, we will present and develop further a technique which deals with the quality improvement of volume rendering, that is pre-integrated volume rendering technique.

Chapter 4

Software

4.1 Introduction

In this chapter, we present the software development environment used in our further implementation: the volume graphics library (VGL). VGL is a C++ class library which enables developers to create rendered images of voxel data in an OpenGL [42] environment combined with OpenGL rendered geometry. We choose VGL as software development environment for our work, due to its rendering facilities and the shortest development and rendering time it provides, in addition to the possibility of creating sample images of any 3D volume data.

4.2 Volumes Representation

Volumes in VGL are represented with the class family "grid of sampled data", where `VGLSampleGrid` (see figure 4.1) is the base class which defines an API to store and access data which is organized as multidimensional array (grid). `VGLSampleGrid` supports up to 4 dimensions.

In our implementation, we use `VGLSampleGrid` and its derived classes namely

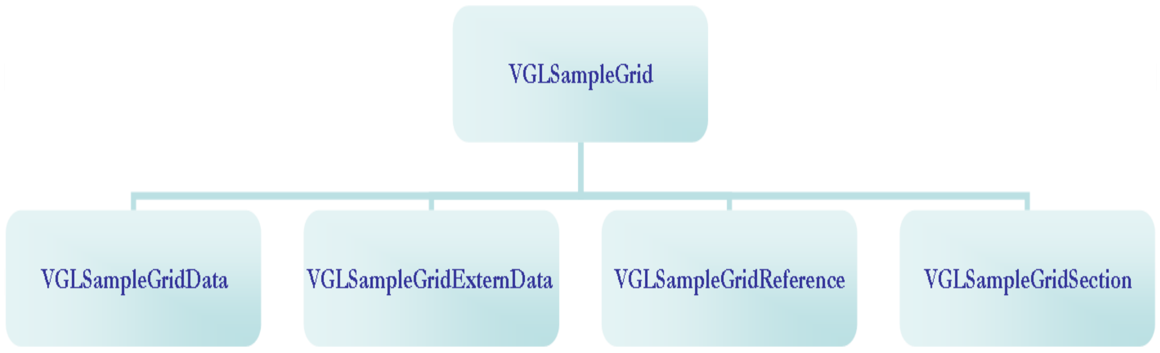


Figure 4.1: VGLSampleGrid Class

`VGLSampleGridData` to store volume data (voxel).

4.3 Rendering Kernel

As shown in figure 4.2, the VGL rendering kernel consist of two distinct parts: first the VGL ray-tracing kernel, and second the VGL direct rendering kernel. They not only differ by their particular rendering approach, but also in their basic characteristics, depending if they are ray-tracing or direct rendering plug-in modules. The rendering kernel is designed to run in an OpenGL environment. VGL needs a valid OpenGL context for rendering in most cases, because the ray-tracing results as well as the direct rendering results will be sent to the OpenGL pipeline.

In the following, we give an overview about both kernels, even we are more interested in the former, since its the one we consider for our development.

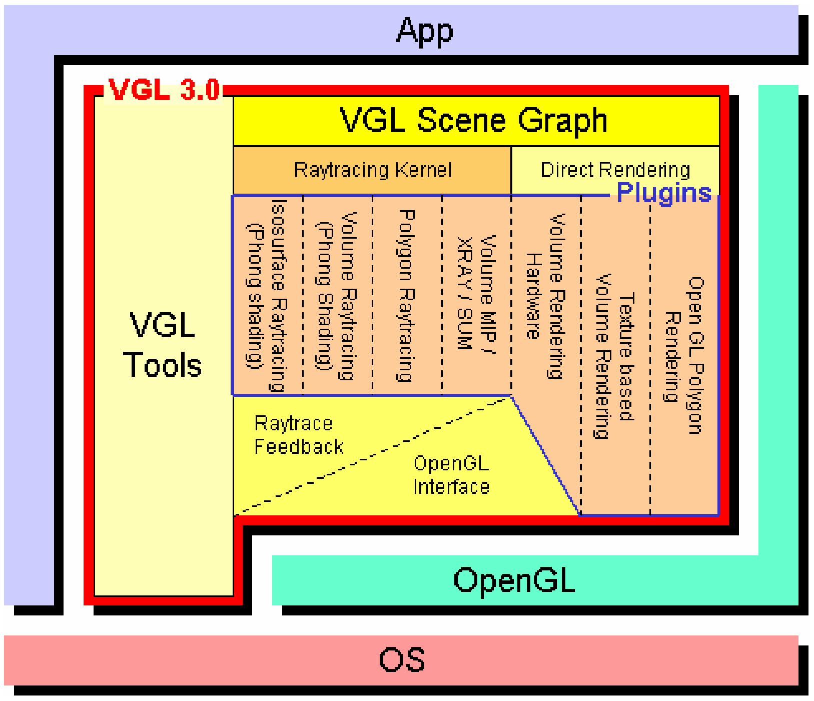


Figure 4.2: VGL rendering kernel

4.3.1 The VGL Ray tracing Kernel

The ray-tracing kernel will be used to render all objects which have one of the ray-tracing plug-in modules installed. The ray-tracing plug-in modules provide the functions to calculate one single ray, and the ray-tracing kernel is responsible to prepare the ray-tracing process (for example, to consider clipping), to schedule the ray-tracing tasks (including multiprocessor support with load balancing), and to send the results to the VGL feedback buffer and to the OpenGL output buffer. The ray-tracing plug-ins have the great advantage that they support all rendering features including all types of light sources, shadow generation, arbitrary clipping, full mixing of all ray-traced objects, and full floating point ray-tracing calculations for best image quality, combined with achievable interactive performance on standard PCs. On the other hand, the ray-tracing plug-ins may be slow, in particular if all sophisticated rendering features are utilized.

4.3.2 The VGL Direct Rendering Kernel

The direct rendering kernel will be used to render all objects which have one of the direct rendering plug-ins installed. Direct rendering plug-ins are responsible for the complete rendering process of a single object, and will send their result to the target OpenGL buffer directly. VGL will call the direct rendering plug-in functions for the appropriate render object either before the ray-tracing kernel starts execution, or after the ray-tracing result has been sent to the OpenGL output buffer. Direct rendering plug-ins have the great advantage that they can utilize special rendering algorithms and hardware support which does not fit into the ray-tracing environment.

4.4 Volume Ray tracing plug-in module

VGL uses plug-in technology at several places. Plug-in modules are dynamically linked libraries (in our case, *.dll files since we work in the Windows XP platform), which are loaded at runtime to provide additional functionality. In this section, we particulary give an overview of the Volume Ray tracing module since we use it further in our implementations.

The volume ray tracing plug-in module can render volumetric data with an arbitrary opacity and color mapping, using a standard sampling-based integration approach. All ray tracing features are supported, including semitransparent volume areas, and correct mixing of semitransparent areas with other ray tracing results. `VGLRayVolPhong` class is the plug-in which implements the ray tracing of volumetric data with a standard sampling based volume rendering approach. For each ray which is requested from the VGL ray tracing kernel, the following tasks will be executed:

1. Upon request from the VGL ray tracing kernel, the next sampling position at the ray will be calculated.

2. Upon request, the Phong lighting equation is integrated along the current interval between two subsequent sampling positions. The calculation includes support for all types of light sources, support for (optional) gradient normalization, and support for shadow generation.

3. The previous steps are repeated until the ray terminates.

4.5 Conclusion

In this chapter, we introduced the VGL class library as background for our development environment, as well as the plug-in module `VGLRayVolPhong` which is the core of the ray tracing module. We further use this plug-in when implementing the preintegration technique, the coherence encoding and the visualization of deformable volumes on ray casting algorithm.

Chapter 5

Pre-Integrated Volume Rendering

5.1 Related Work

5.1.1 Volume Rendering Integral

The transport theory of light [30, 50, 32] is the basis for many volume rendering methods and results in the following volume rendering integral:

$$I = \int_0^B q(x(t))\ e^{-\tau(0,t)} dt. \qquad (5.1.1)$$

I is the intensity at the position B (B is the location of the background), along a ray $x(t)$, in three-dimensional space . With q being the reflection function and τ, the *optical depth* defined as

$$\tau(t_1, t_2) = \int_{t_1}^{t_2} \kappa(t)\ dt, \qquad (5.1.2)$$

where κ is the *opacity function*. For numerical evaluation, this integral defined on the interval $[0, B]$ is further subdivided into small not necessarily equidistant subintervals $[t_k, t_{k+1}],\ k = 0, \ldots, N-1$, where $t_0 = 0$ and $t_N = B$. This results in the well known recursive compositing formulae:

$$\begin{aligned} \tilde{C}_k &= \tilde{C}_{k-1} + (1 - \tilde{\alpha}_{k-1})\ C_k, \\ \tilde{\alpha}_k &= \tilde{\alpha}_{k-1} + (1 - \tilde{\alpha}_{k-1})\ \alpha_k, \qquad k = 1, \ldots, N-1, \end{aligned} \qquad (5.1.3)$$

where $\tilde{C}_k$ is the *pixel color*, and the *voxel color* C_k is defined as:

$$C_k = \int_{t_k}^{t_{k+1}} q(t)\ e^{-\tau(t_k,t)} dt, \tag{5.1.4}$$

Further, $\tilde{\alpha}_k$ is the *pixel opacity*, and the *voxel opacity* α_k is

$$\alpha_k = 1 - e^{-\tau(t_k,t_{k+1})}. \tag{5.1.5}$$

Finally $\tilde{C}_0 = C_0, \tilde{\alpha}_0 = \alpha_0$ and $\tilde{C}_{N-1} = I$.

5.1.2 Prior work about pre-integration

Different strategies for numerically solving the volume rendering integral (5.1.1) were discussed by Max [40]. His idea was to precompute and tabulate the indefinite rendering integral between values defined at the vertices of the polyhedra for the projection method, assuming that the density scalar function varies linearly along the ray. Improvements were further discussed by Williams et al [61]. They calculated the volume rendering integral cell by cell assuming that the transfer function varies piecewise linearly along a ray segment within each cell. Later, simplifications were derived by Stein et al [62]. They developed a faster but less accurate method, where they assumed the opacity varies linearly along the ray segment and assumes the color is constant equal to the average of the color at the front and the back of the ray segment. This was not correct, since the opacity along a ray segment hides the far color more than the near one, but was much quicker to evaluate. Roettger et al [49] applied this technique as an enhancement to the Projected Tetrahedra algorithm using 3D texture hardware for enabling the use of arbitrary transfer functions and for rendering isosurfaces without reconstructing them geometrically. Compared to Stein, Roettger et al developed a scheme to approximate (5.1.1) for different sampling distances which allows to pre-integrate the contribution of the volume rendering integral within the considered interval by a 2D LUT instead of a 3D LUT.

Engel et al [10] named this technique pre-integration and extended it by computing the colors in the same way as Roettger et al but considering their modulation by diffuse and specular shading. For the normals, they average the normals of the two subsequent sampling points under consideration. However, to accelerate the computation of the pre-integration tables, they hold the distance between sample points constant and neglect the attenuation within the segment intervals. Later improvements [47] consider an optimized LUT-generation and final rendering using 2D texture hardware. Later, Roettger et al [48] improved the algorithm by super-sampling and accurate clipping in order to achieve a better image quality. An application of pre-integration to Shear Warp type algorithms was suggested by Schulze et al [52], they rendered slabs between adjacent slices instead of individual slices using a buffer slice to store interpolated scalar values of the back slice, and stored the result into a 2D lookup table considering a constant distance d. They improved image quality at the expense of the algorithm performance. Lum et al [38] improved the pre-integration for lighting by linearly interpolating the lighting in the front and back sample points which is a correct method for diffuse shading but only an approximation for the specular contribution. Further suggestions were given for computing the pre-integration LUTs. [41] restricted to piecewise linear transfer functions which allows to precompute a pre-integration tables being independent on the classification. The approaches so far approximate the volume rendering integral by simplifying the shading, partially ignoring self-shadowing, and assuming linear changes of the scalar function between two sample points.

In the following, a comprehensive approach is given which describes the precomputation without simplifications and where the normals depend on the opacity instead of the gray values. We further show that for linear, quadratic, and higher-order polynomial models, for the change of the gray values between two sample points, a pre-integration on a set of 2D LUTs is possible, which can be computed in a short time. Further, numerical analysis give indicators for finding suited sizes of the LUTs.

5.1.3 Basic Algorithm

The form of the volume rendering integral (5.1.1) is not useful for the visualization of a continuous scalar field $g(\vec{x})$, because the calculation of the color and opacity coefficients is not specified. This calculation includes two main calculation steps. First, the classification which maps the scalar values $g = g(\vec{x})$ to color function $c(g)$ and opacity function $\kappa(g)$. The second step is the shading calculation, i.e, the color contribution of a point in space. In [9], Engel assumed the scattering part in equation (5.1.1) depending on the color classification function $c(g)$. Therefore, the volume rendering integral can be rewritten as:

$$I = \int_0^B c(g(\vec{x}(t)))\ exp(-\int_0^t \kappa(g(\vec{x}(t')))\ dt')dt. \tag{5.1.6}$$

A numerical integration is required to evaluate this volume rendering integral. The most common one is the calculation of a Riemann sum for N equal ray segments of length $D = B/N$ (see figure 5.1). This approximation in volume rendering integral is not sufficient when considering complex transfer function, and results in artifacts due to the problem of high Nyquist frequencies of $c(g(\vec{x}))$ and $\kappa(g(\vec{x}))$. To overcome this limitation, Engel et al [9] split the pre-integration classification into two integrations: one for the continuous scalar field $g(\vec{x})$ and one for the transfer functions $c(g)$ and

$\kappa(g)$. The Nyquist frequency for the sampling of the continuous scalar field $g(\vec{x})$, along a viewing ray, is not affected by the transfer functions. As shown in figure 5.1, for the pre-integrated classification, the sampled values define a one dimensional, piecewise linear scalar field. The volume rendering integral for this piecewise linear scalar field is efficiently computed by one table lookup for each linear segment. This lookup table depends on three parameters:

1. The scalar value at the entry of the sampling interval, $S_f = g(\vec{x}(t_k))$,
2. The scalar value at the end of the sampling interval, $S_b = g(\vec{x}(t_{k+1}))$,
3. The sampling interval length $D := t_{k+1} - t_k$.

Thus, the opacity function α_k and the color function c_k, of the $k-th$ segment can be respectively written by:

$$\begin{aligned} \alpha_k &= 1 - exp(-\int_{t_k}^{t_{k+1}} \kappa(g(\vec{x}(t)))dt) \\ &\approx 1 - exp(-D\int_0^1 \kappa((1-w)S_f + wS_b)dw) \end{aligned} \tag{5.1.7}$$

and

$$\begin{aligned} c_k &= \int_{t_k}^{t_{k+1}} c(g(\vec{x}(t))) \\ &* \quad exp(-\int_{t_k}^{t} \kappa(g(\vec{x}(t')))dt')dt \\ &\approx D\int_0^1 c((1-w)S_f + wS_b) \\ &* \quad exp(-D\int_0^w \kappa((1-w')S_f + w'S_b)dw')dw \end{aligned} \tag{5.1.8}$$

Thus, pre-integrated classification allows sampling a continuous scalar field $g(\vec{x})$ without the need to increase the sampling rate for any non-linear transfer function.

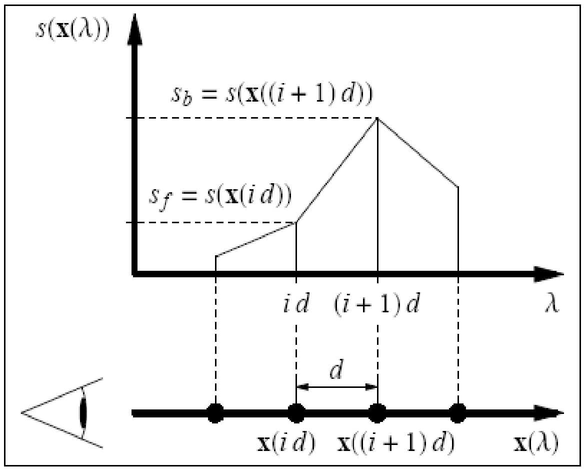


Figure 5.1: Scheme of the parameters determining the color and opacity of the i-th ray segment.(By Engel [9])

Finally, to accelerate pre-integrated lookup table generation, Engel et al [9] assumed a constant sampling distance D, and neglect the attenuation within a ray segment. These simplifications lead to the following equations of the opacity and color function:

$$\alpha(S_f, S_b, D) = 1 - exp(-\frac{D}{S_b - S_f}\int_{S_f}^{S_b} \kappa(s))ds) \tag{5.1.9}$$

and

$$c(S_f, S_b, D) = \frac{D}{S_b - S_f}\int_{S_f}^{S_b} c(s)ds \tag{5.1.10}$$

In the following, we describe how pre-integrated volume rendering can be realized using the correct non-simplified volume rendering integral, i.e., considering the attenuation factor as well as the shading function during the precomputation process. In the first part, we assume that the gray values vary linearly between sample points along the rendering interval, considering the ambient, diffuse and specular reflections respectively. Then, an approach to deal with different sampling distances is shown and finally we describe the details of implementation of the LUTs for this linear model. In the second part, we extend our method to quadratic models and finally to cubic and higher order polynomials. In the remainder of this section, we assume the scattering part in equation (5.1.4), depending on the color classification function $c(g)$ and the gradient of the opacity classification function $\kappa(g)$ where $g(\vec{x}(t))$ is the gray value of the point $\vec{x}$ at position t of the ray. We further denote $g(\vec{x}(t))$ by $g(t)$. Thus, $q(t) := q(c(g(t)), \vec{\nabla}(\kappa(g(t))))$, whereby q is any reflection function, in our case the Phong illumination model [44]. We also assume the opacity function in equation (5.1.2), depending on $g(t)$, i.e., $\kappa(t) := \kappa(g(t))$.

5.2 Linear Gray Value Model

The linear model, where the gray values between two sampling points change linearly, is the most simple model describing a sampled scalar function. For rendering, we consider the Phong illumination model for shading, where the gradients are computed from the *opacity function* κ. In this section, we will first calculate the opacity function along a sampling interval, then we will evaluate all three components of the Phong model, separately.

5.2.1 Opacity Evaluation

We first remind the expression of the opacity integral, within a sampling interval of length D:

$$\alpha = 1 - exp(-\int_0^D \kappa(g(t))dt) \qquad (5.2.1)$$

A reformulation of the gray values function $g(t) = t(S_b - S_f)/D + S_f$, as well as a variable substitution $s := g(t)$ lead to the following expression of α:

$$\begin{aligned} \alpha &= 1 - exp(-\frac{D}{S_b - S_f}\int_{S_f}^{S_b} \kappa(s)ds) \\ &= 1 - exp(-\frac{D}{S_b - S_f}(\int_{0}^{S_b} \kappa(s)ds - \int_{0}^{S_f} \kappa(s)ds)) \end{aligned} \quad (5.2.2)$$

To speed up the evaluation of this expression, we first precompute, for each gray value $s \in [0, N-1]$, the integral $A(s) = \int_0^s \kappa(s')ds'$. This integral is easily computed in practice, as the scalar values s are usually quantized. The result is therefore, stored into a 1D LUT of size equal to 32 $bits \times 256 = 1KB$. Thus, the expression of α can be written as:

$$\alpha = 1 - exp(-\frac{D}{S_b - S_f}(A(S_b) - A(S_f))) \quad (5.2.3)$$

The pseudo-code of the function calculating A is given in the algorithm 1.

Algorithm 1 A(a,b): Accumulated opacity

```
$A(0) = 0$;
**for** $s = 1$ to $N$ **do**
  $A(s) = A(s-1) + (a[s-1] * s + b[s-1] - 0.5 * a[s-1])$
**end for**
```

In the pseudo-code of A, $a[s-1]$ and $b[s-1]$ are the coefficients of the piecewise linear opacity function within the interval $[s-1, s]$. They are obtained from the opacity TF values at $s-1$ and s. Finally, for each distance D, we compute α between S_f and S_b and store it into a 2D LUT. The pseudo-code of the algorithm calculating the opacity α is described by the algorithm 12 (see Appendix A).

5.2.2 Ambient Reflection

The ambient contribution of the volume rendering integral between two sample points including self-shadowing and having a linear gray value model for the scalar function reads:

$$C_a = K_a \int_0^D c(g(t))\ exp(-\tau(0,t))dt, \tag{5.2.4}$$

where K_a is the ambient reflection coefficient, D is a constant sampling distance (i.e., $D := t_{k+1} - t_k$, where t_k is the parameterized location of the sample point). Let us reformulate this integral by formulating the linear function $g(t)$ as a function of the gray values of the interval boundaries $S_f,\ S_b$ at t_{k+1} and t_k as:$g(t) = t(S_b - S_f)/D + S_f$. Using the variable substitution $s := g(t)$, we obtain:

$$\begin{aligned} C_a &= K_a d\ exp(\omega(d, S_f)) \int_{S_f}^{S_b} c(s)\ exp(-\omega(d,s))ds \\ &= K_a d\ exp(\omega(d, S_f)) \sum_{i=S_f}^{S_b-1} \int_i^{i+1} c(s)\ exp(-\omega(d,s))ds \end{aligned} \tag{5.2.5}$$

where

$$d = \frac{D}{S_b - S_f}.$$

and

$$\omega(d,s) = d\int_0^s \kappa(s\prime)ds\prime.$$

In the original pre-integrated technique, the attenuation within a ray segment was neglected [10]. In our approach, we do not consider this approximation. Our idea is to generate the final 2D LUT from precomputed 1D LUT. Thus, let us consider for each gray value $i \in [0, N-1]$ and a given value of d the function C_a defined by:

$$C_a(i,d) = \int_i^{i+1} c(s)\ exp(-\omega(d,s))ds. \tag{5.2.6}$$

To evaluate C_a, we will first calculate $C_a(i,d)$ for each i value and store the results into a lookup table of entries i and d. From these LUTs, a 2D LUT depending on the sampling distance D and having as entries S_f and S_b is generated. In case where S_f and S_b are not whole numbers, we perform, during rendering, a bilinear interpolation between the four nearest neighbors of a non-integer gray value, within the LUTs.

5.2.3 Diffuse Reflection

Now, let us discuss the diffuse reflection. The respective part of the rendering integral is:

$$C_d = K_d \int_0^D c(g(t))(\vec{\nabla}\kappa(g(t)) \cdot \vec{L})\ exp(-\tau(0,t))dt, \qquad (5.2.7)$$

where "$\cdot$" denotes the scalar product. If the expression in the parenthesis can be computed via a LUT, we obtain a vector that only needs to be multiplied by the light direction $\vec{L}$ which will be evaluated during shading and thus the result can be computed directly. Unfortunately, this expression not only depends on the front and back gray value, which would lead to a 2D LUT, but it also depends on the gradient of the opacity function. Earlier versions used not the opacity but the gray value and there the gradient direction remains constant and thus simplifies the situation. However, this choice has practical disadvantages like incorrect shading results when choosing arbitrary opacity functions. Therefore, we still remain on the physical basis that the normal of a surface depends on the opacity function. Let us try to evaluate the gradient $\vec{\nabla}\kappa(g(t))$ as:

$$\begin{aligned}\vec{\nabla}\kappa(g(t))_{i=1..3} &= (\frac{\partial\kappa}{\partial g}\frac{\partial\vec{g}}{\partial t_i})(g(t)) \\ &= \frac{\partial\kappa}{\partial g}(\frac{\partial\vec{g}}{\partial t})_i(g(t)) \\ &= \frac{\partial\kappa}{\partial g}(g(t))\cdot\vec{\nabla}g \\ &= \frac{\partial\kappa}{\partial g}(g(t))\cdot\vec{N} \qquad (5.2.8)\end{aligned}$$

Thus, we can rewrite the equation (5.2.7) as:

$$C_d = K_d(\vec{N}\cdot\vec{L})\int_0^D c(g(t))\frac{\partial\kappa}{\partial g}(g(t))\ exp(-\tau(0,t))dt. \qquad (5.2.9)$$

$\vec{N}$ is constant over the integration area since g is linear. Its value is determined during rendering. The same variable substitution cited in the ambient reflection transforms the above equation (5.2.9) to:

$$\begin{aligned}C_d &= K_d d\ (\vec{N}\cdot\vec{L})\ exp(\omega(d,S_f))\int_{S_f}^{S_b} c(s)\frac{\partial\kappa}{\partial s}(s)\ exp(-\omega(d,s))ds. \\ &= K_d d\ (\vec{N}\cdot\vec{L})\ exp(\omega(d,S_f))\sum_{S_f}^{S_b}\int_i^{i+1} c(s)\frac{\partial\kappa}{\partial s}(s)\ exp(-\omega(d,s))ds \qquad (5.2.10)\end{aligned}$$

As explained for the ambient reflection, we first precalculate the 1D LUTs that store for every gray value $i\in[0,N-1]$:

$$C_d(i,d) = \int_i^{i+1} c(s)\frac{\partial\kappa}{\partial s}(s)\ exp(-\omega(d,s))ds. \qquad (5.2.11)$$

The difference with $C_a(i,d)$ is that we multiply by the term $\frac{\partial\kappa}{\partial s}(s)$. Since we assume the opacity transfer function being piecewise linear, i.e., the opacity function being linear between two consecutive gray values, $\frac{\partial \kappa}{\partial s}(s)$ is therefore constant for each $s \in [s_i, s_{i+1}]$. According to this, we can generate the 1D LUTs without problems. The diffuse 2D LUTs are finally generated as described for the ambient reflection. During shading, we have only to lookup the 2D diffuse LUTs, to deduce the corresponding integral value of the current sampling interval and multiply it on the fly by the scalar product between $\vec{N}$ and $\vec{L}$.

5.2.4 Specular Reflection

Finally, let us discuss the case for specular reflection. The respective part of the ray integral is:

$$C_s = K_s \int_0^D c(g(t))(\vec{\nabla}\kappa(g(t)) \cdot \vec{H})^{n_\delta} \ exp(-\tau(0,t))dt. \tag{5.2.12}$$

where $\vec{H}$ is the halfway vector and n_δ is the specular exponent. Given the notations mentioned before, we transform the above formula to:

$$C_s = K_s(\vec{H} \cdot \vec{N})^{n_\delta} \int_0^D c(g(t))(\frac{\partial \kappa}{\partial g}(g(t)))^{n_\delta} \ exp(-\tau(0,t))dt \tag{5.2.13}$$

After variable substitution, we arrive at:

$$C_s = K_s(\vec{H} \cdot \vec{N})^{n_\delta} d \ exp(-\omega(d, S_f)) \int_{S_f}^{S_b} c(s)(\frac{\partial \kappa}{\partial s}(s))^{n_\delta} \ exp(-\omega(d,s))ds \tag{5.2.14}$$

As for the diffuse reflection, we precompute the 1D LUTs with a single difference which is storing $(\partial\kappa/\partial s(s))^{n_\delta}$ instead of $(\frac{\partial \kappa}{\partial s}(s))$. Thus, the equation (5.2.14) can be evaluated by first computing the following integral:

$$C_s(i,d) = \int_i^{i+1} c(s)(\partial\kappa/\partial s(s))^{n_\delta} \ exp(-\omega(d,s))ds. \tag{5.2.15}$$

Then, the equation (5.2.14) can be stored (up to $\vec{N}$ and $\vec{H}$) into 2D LUTs as before. We notice that a change on the specular exponent n_δ leads only to update the specular

2D LUTs, without any need to precompute the tables related to ambient and diffuse reflection.

5.2.5 Integration Length

Up to this point, we have seen that the Phong shading approach is compatible with the pre-integration technique, using as many 1D LUTs as necessary to generate 2D LUT for every reflection part of the shading illumination model. There is therefore only one free parameter that has not been considered yet, i.e., the sampling distance D. The volume rendering integral (5.1.1) can be rewritten for $D = 1$, by a variable substitution $z = t/D$:

$$I = D\left[\int_0^1 q(c(h(z)), \vec{\nabla}\kappa(h(z)))\ exp(-D\int_0^z \kappa(h(z'))dz')dz\right] \tag{5.2.16}$$

where $h(z) = g(Dz)$. As can be seen, this substitution does not help much since the scaling factor D also appears in the exponential. However, we could consider the class of integrals:

$$T = \int_0^1 q(c(h(z)), \vec{\nabla}\kappa(h(z)))\ exp(-D\int_0^z \kappa(h(z'))dz')dz. \tag{5.2.17}$$

Earlier papers considered the absorption part only after having computed the contribution for the considered integral and therefore the length of the integral had no consequence on the computation. In our case, however, we want to consider the length of the integral as well. Since there is no linear and no polynomial relationship of the integral value I on D, the only way to overcome this problem is by storing for different integration lengths as well, increasing the size of the LUTs. However, since the D-fold exponential factor leads only to subtle changes, only a few sample values are necessary whereas the rest can be obtained by dedicated interpolation. This is done by a logarithmic mapping of the thickness D, as suggested by Kraus in [31]. The dimension of the LUT in $\vec{D}$ direction is decomposed into intervals $[d_j, d_{j+1}]$ such as $d_j := 2^j$ and $j \in [j_{min}..j_{max}]$. Thus, the values of d_j cover the whole range of $[0, D]$. So, for each d_j, a 1D LUT for $C(i, d_j), i \in [0, N-1]$ is generated, where N is the LUT size. Another alternative would be to compute LUTs for small intervals only, $D < 1$. The size of the intervals is chosen such that a linear or spline interpolation between them gives accurate enough results for any D in between the discrete choices, then, we can handle large D values by compositing a number of small intervals one after the other.

5.2.6 Realization

Earlier in this section, we introduced the strategy for generating the lookup tables to store the opacity and reflection values for Phong shading. Now, we will describe how to realize that. The first step to generate the lookup tables, was to precalculate 1D LUT to store respectively, for the ambient, diffuse and specular reflection the values of the equations (5.2.6), (5.2.11) and (5.2.15). For simplicity, we will consider for this section, the case of the diffuse reflection lookup table generation. The other reflection types can be obtained by deduction. For this, let us consider the equation (5.2.11). Since $C_d(i, d)$ is a continuous function, a dedicated sampling for generating a suited LUT is required. Since we assumed the color transfer function being piecewise linear, the Trapezoid rule for numerical integration is then a good choice to approximate $C_d(i, d)$. In equation (5.2.11), let:

$$f_d(s) = c(s)\frac{\partial \kappa}{\partial s}(s)\ exp(-\omega(d, s)), s \in [s_i, s_{i+1}]. \tag{5.2.18}$$

$c(s)$ is obtained by linear interpolation between $c(s_i)$ and $c(s_{i+1})$. $\frac{\partial \kappa}{\partial s}(s)$ is constant within the interval $[s_i, s_{i+1}]$, since the opacity transfer function is assumed to be piecewise linear. Finally, the value of $exp(-\omega(d,s))$ is obtained by:

$$exp(-\omega(d,s)) = 1 - LIN(opacity[j][0][s_i], opacity[j][0][s_{i+1}], h) \tag{5.2.19}$$

where *opacity* is the already generated opacity LUT, and j the corresponding index for the distance d. LIN here, designs the linear interpolation function. Once the 1D LUT are generated, we can deduce the final 2D LUT by using an incremental algorithm. The details of the implementation of this algorithm are described in the following section. Later, we refer to the 1D LUT as $color1d$.

5.2.6.1 Incremental Algorithm for Generating LUT for Linear Model

The goal of our algorithm is to generate, for the different sampling distances we considered, 2D LUT which store for each couple of gray values S_f and S_b, at the entry and exit point of a sampling interval, the reflection contribution in between. As we mentioned in the section 5.2.5, let decompose the range $[0, D]$ in intervals $[d_j, d_{j+1}]$ such as: $d_j := 2^j$ and $j \in [j_{min}..j_{max}]$. First, Let us remaind the equation (2.3.17) in chapter 2:

$$I(s_i) = I(s_{i-1})e^{-\tau(s_{i-1},s_i)} + \int_{s_{i-1}}^{s_i} q(s)e^{-\tau(s,s_i)}ds \tag{5.2.20}$$

By introducing the substitutions given in equations (2.3.18) and (2.3.19), the equation (5.2.20) can be rewritten by:

$$I(s_i) = I(s_{i-1})\theta_i + b_i \tag{5.2.21}$$

Interpreting in equation (5.2.21):

- θ_j and b_j as being respectively, the transparency and the color in between the gray value interval $[g-1, g]$
- $I(s_{i-1})$ as $C_{j-1}(g-1)$: the reflected intensity (here shading color) at distance d_{j-1}
- and $I(s_i)$ as $C_j(g)$: the reflected intensity at distance d_j,

one can write the following recurrence relation:

$$C_j(g) \;:=\; \begin{cases} 0 & \text{if} j < j_{min} \\ C_{j-1}(g-1)\theta_j + b_j & \forall \mathrm{j} \in [\mathrm{j_{min}..j_{max}}] \end{cases} \tag{5.2.22}$$

First, we initialize all reflection values for $d = 0$ to *zero*. Then, we start generating the 2D LUT in d direction for $S_b = S_f$. For this case, the results are generated analytically as mentioned before and are stored into a temporary array $Temp$ of size $j_{max} - j_{min} + 1$. Therefore, we keep S_f constant, increment S_b and repeat the process until $S_b = N$ (N the total size of the 2D LUT). Finally, we increment S_f and the same procedure is reiterated until $S_f = N$. Given the recurrence relation (5.2.22), we generate all the reflection values $C_j(b)$ for $j \in [j_{min}..j_{max}]$. The calculation of $C_j(b)$ involves the evaluation of:

1. θ_j obtained from the opacity LUT at d_j between S_{b-1} and S_b by:

$$\theta_j = 1 - opacity[j][b-1][b] \tag{5.2.23}$$

2. b_j read from the already precomputed 1D LUT *color*1*d* described in the previous section.

$$b_j = color1d[j][b] \tag{5.2.24}$$

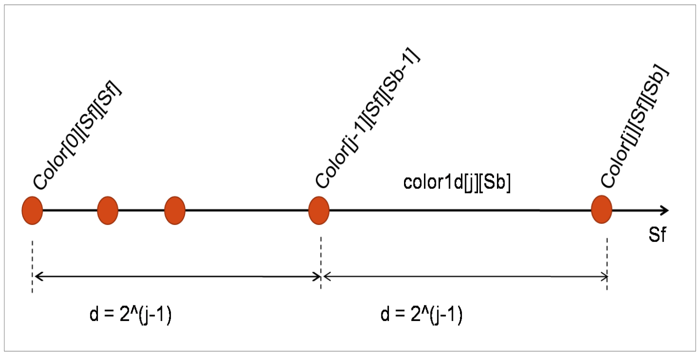


Figure 5.2: Incremental calculation of the color by compositing

3. $C_{j-1}(g-1)$ read from the temporary array $Temp$.

Finally, we save the calculated value $C_j(g)$ on $Temp[j+1]$, as shown in figure 5.2. A pseudo-code of our incremental algorithm is given in algorithm 2.

Algorithm 2 Get Linear Color: Incremental algorithm for generating lookup tables when the gray value model is linear

```
for $S_f = 0$ to $N$ do
  for $S_b = S_f$ to $N$ do
    if $S_f = S_b$ then
      for $j = jmin$ to $jmax$ do
        $Evaluate\ analytically\ color([j][S_f][S_b])$;
        $Temp[j+1] = color[j][S_f][S_b]$;
      end for
    else
      for $j = jmin$ to $jmax$ do
        $color[j][S_f][S_b] = composite(color1d[j][b], Temp[j+1])$;
        $Temp[j+1] = color[j][S_f][S_b]$;
      end for
    end if
  end for
end for
```

5.2.6.2 Algorithmic Complexity

Let n_d being the number of distances interfering with the color LUT generation of size $n \times n$. The above described algorithm requires, for each axis d_j, n_d multiplications and additions. Thus, along the axis S_b, the number of operations needed is: $n_d \times i$ such as: $S_b \in [n-i, n]$ with $i \in [0..n-1]$. Finally, along the axis S_f, the number of operations required is: $\sum_{i=0}^{n-1} n_d i = n_d \frac{n(n-1)}{2} \sim n^2$. Therefore, the complexity of our incremental algorithm is $O(n^2)$. Thus, our algorithm is better than the $O(n^3)$ Lum's algorithm [38]. In addition, we perform the computations for different interval lengths. Thus, we addressed the problem of table generation speed. It is important that the computation of the lookup tables be efficient so that the transfer function may be interactively modified.

5.3 Quadratic Case

5.3.1 Motivations

Most techniques for the visualization of volume data require appropriate modeling. Exact analytic solution cannot be calculated, but approximated. The pre-integration technique has eliminated many of the rendering artifacts that arise from slicing the volume but it assumes a linear progression of the scalar values between two sample points along the ray. This assumption does not match the quadratic behavior of the scalar values. For this reason, we propose to extend the pre-integration technique for quadratic polynomials. In this section, we present a new model of reconstruction of volume data, namely the quadratic model: we approximate the density function by a piecewise quadratic polynomial. Therefore, we can expand the pre-integration technique, for $g(t)$ being a second degree polynomial, i.e, $g(t) = at^2 + bt + c$. In addition, we select the opacity function and the color transfer function being piecewise quadratic.

5.3.2 Strategy

Our goal is to precompute opacity and reflection values for quadratic density model, and store the results into a LUT that can be updated in short time. Unlike the linear case, the volume rendering equation is now depending on one more parameter. This is due to the three coefficients present in the density function. Thus, the volume rendering expression depends on four parameters a, b, c and D. This would lead to a four dimensional lookup table, which is computationally expensive and memory consuming. Hence, the idea is to decrease the dimension of the LUT by minimizing the number of involved parameters. We address this problem by performing a variable substitution to reduce the number of parameters, while using the same strategy used for the linear model to deal with the distance length parameter. In the next section, we describe how is that possible. We start by the opacity calculation, then the reflection evaluation for Phong shading.

5.3.3 Opacity Calculation

The opacity function α is defined as:

$$\alpha = 1 - \exp(-\int_0^D \kappa(g(t))dt). \tag{5.3.1}$$

In the following, for simplicity, we only develop the case where $a > 0$. The case where $a < 0$ can be deduced by simple analogy. Under this assumption, the equation (5.3.1) can be rewritten, by a variable substitution $z = t/R$, as:

$$\alpha = 1 - \exp(-R\int_0^{D'} \kappa(h(z))dz). \tag{5.3.2}$$

where $R = 1/\sqrt{\mid a \mid}$ and $D' = D/R$. By noting $b' = bR$, $h(z)$ is such as: $h(z) = z^2 + b'z + c$. The dependency of $h(z)$ on R is problematic. Let us discuss the issues to this problem according to the values of a. First, we choose a threshold ε so that: if $a < \varepsilon$, the opacity value is obtained by looking up into the 2D LUT opacity generated for the linear case, otherwise, we have to evaluate the equation (5.3.2). Let define β being:

$$\beta = \int_0^{D'} \kappa(h(z))dz \tag{5.3.3}$$

Therefore, evaluating (5.3.2) means calculating β. Applying directly the variable substitution $s = h(z)$ to the integral β leads to wrong results because of the lacking bijectivity of h due to the possible existence of an extremum in $h(z)$. Let denote this extremum if it exists by $M = h(-b'/2) = c - b'^2/4$. Therefore, depending on the value of b', we distinguish three possible cases for computing β: $\begin{cases} b' \geq 0; \\ b' \in]-2D', 0[; \\ b' \leq -2D'. \end{cases}$

5.3.3.1 Case I: $b' \geq 0$

As shown in the figure 5.3, h is monotonous and then a variable substitution $s = h(z)$ can be applied, we obtain therefore:

$$\beta = \int_{S_f}^{S_b} \frac{\kappa(s)}{2\sqrt{s-M}}ds = \sum_{i=S_f}^{S_b-1} \int_i^{i+1} \frac{\kappa(s)}{2\sqrt{s-M}}ds \tag{5.3.4}$$

Further, β_i is defined as:

$$\beta_i = \int_i^{i+1} \frac{\kappa(s)}{2\sqrt{s-M}}ds \tag{5.3.5}$$

We evaluate the term β_i by the Simpson rule, since we are considering that the opacity varies piecewise quadratically inside the sampling interval. Thus, we need three values to approximate the opacity function in this region: $\kappa(i-0.5), \kappa(i)$ and $\kappa(i+0.5)$.

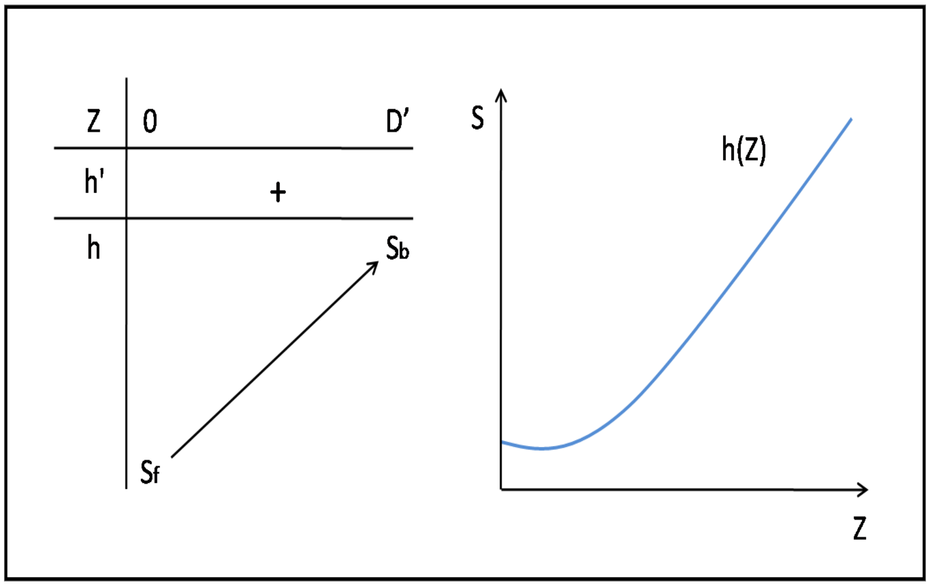


Figure 5.3: Variation of h when $b' \geq 0$

5.3.3.2 Case II: $b' \in]-2D', 0[$

For this case, as shown in figure 5.4, β has to be divided into two parts:

$$\beta = \int_0^{-b'/2} \kappa(h(z))dz + \int_{-b'/2}^{D'} \kappa(h(z))dz \tag{5.3.6}$$

The variable substitution $s = h(z)$ leads for the first subintegral, to $z = \frac{-b'-2\sqrt{s-M}}{2}$ and for the second subintegral, to $z = \frac{-b'+2\sqrt{s-M}}{2}$. Now, β can be rewritten as:

$$\begin{aligned} \beta &= \int_{S_f}^{M} \frac{\kappa(s)}{-2\sqrt{s-M}}ds + \int_{M}^{S_b} \frac{\kappa(s)}{2\sqrt{s-M}}ds \\ &= \int_{M}^{S_f} \frac{\kappa(s)}{2\sqrt{s-M}}ds + \int_{M}^{S_b} \frac{\kappa(s)}{2\sqrt{s-M}}ds \\ &\simeq \sum_{i=E(M)}^{S_f-1} \beta_i + \sum_{i=E(M)}^{S_b-1} \beta_i \end{aligned} \tag{5.3.7}$$

Where $E(M)$ is the ceiling function of the variable M, because i should be a discrete number. The first two terms of this equation are computed in the same way as

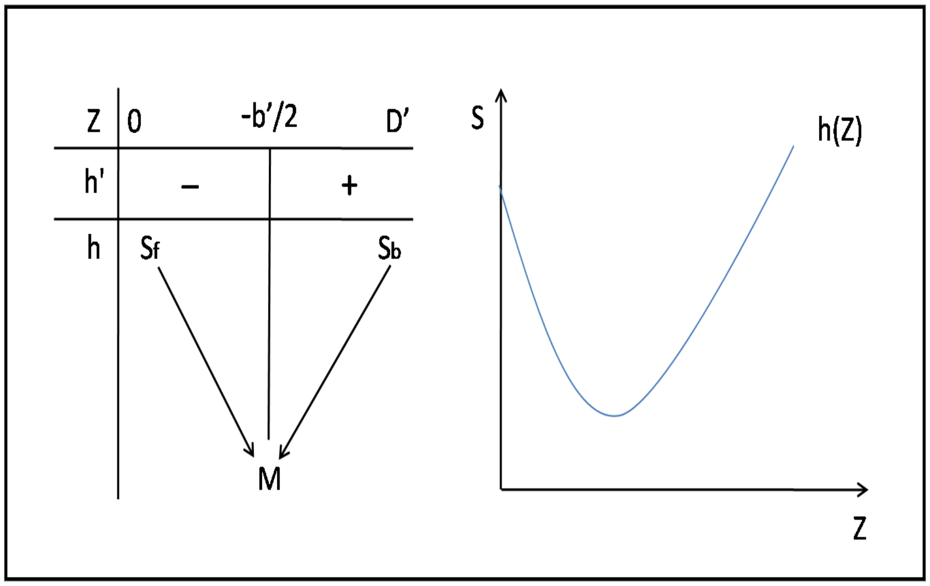


Figure 5.4: Variation of h when $b' \in]-2D', 0[$

described for *case I*.

For the case where $M < 0$, and since $\kappa(s)$ is not defined for $s < 0$, we solve this problem by setting the negative part of the curve to zero. β is then equivalent to:

$$\beta = \sum_{i=0}^{S_f - 1} \beta_i + \sum_{i=0}^{S_b - 1} \beta_i \tag{5.3.8}$$

5.3.3.3 Case III: $b' \leq -2D'$

As shown in the figure 5.5, h is also monotonous and then a variable substitution $s = h(z)$ can be applied, we obtain therefore:

$$\beta = \int_{S_f}^{S_b} \frac{\kappa(s)}{-2\sqrt{s-M}} ds \quad = \quad -\sum_{i=S_f}^{S_b - 1} \int_i^{i+1} \frac{\kappa(s)}{2\sqrt{s-M}} ds \tag{5.3.9}$$

We notice that this equation is no other than the equation 5.3.4 multiplied by -1. For all the cases described above, the value of the opacity α is defined such as:

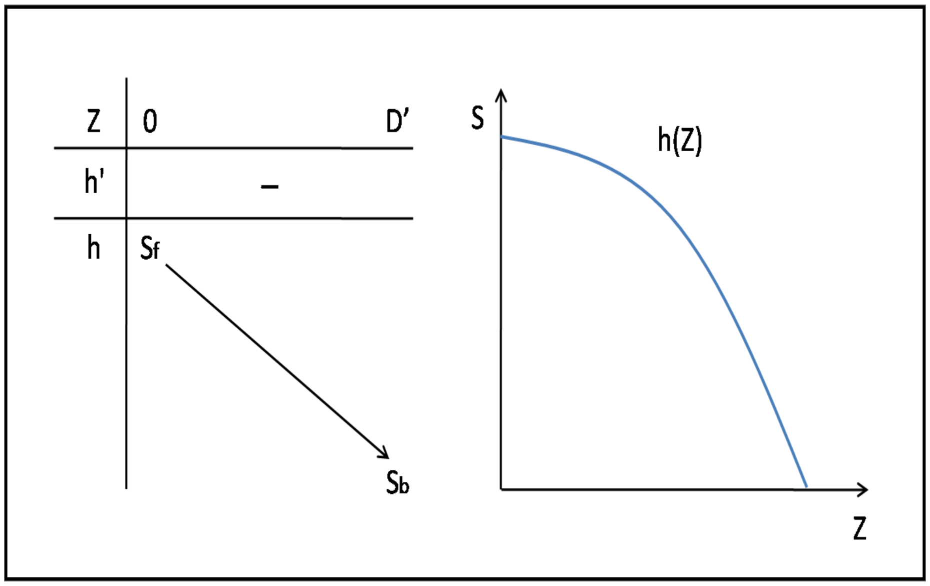


Figure 5.5: Variation of h when $b' \leq -2D'$

$$\alpha = 1 - \exp(-R\beta). \tag{5.3.10}$$

Since β depends on the parameters S_f, S_b and M. Or,

$$\begin{aligned} M &= c - b'^2/4 \\ &= S_f + \frac{1}{4}(\frac{S_b - S_f - D'^2}{D'}) \end{aligned} \tag{5.3.11}$$

Thus, α depends on S_f, S_b, D' and R. Therefore, to store the opacity values, we will create a 2D LUT depending on D' and R. For this reason, the D' and R range have to be discretized. As described in the last section, for the distance length, we use a logarithmic mapping to cover the range of these both parameters. S_f and S_b are chosen to be whole numbers, and in case where not, we perform during rendering, a bilinear interpolation between the four nearest neighbors of a non-integer gray value, within the LUTs.

5.3.4 Reflection Calculation

Under the same notations and by a variable substitution $z = t/R$, equation (5.1.1) can be rewritten, at $t = D$, as:

$$I = R\left[\int_0^{D\prime} q(c(h(z)), \vec{\nabla}\kappa(h(z)))\ exp(-R\int_0^z \kappa(h(z'))dz')dz\right]. \qquad (5.3.12)$$

By applying the variable substitution $s = h(z)$ to equation (5.3.12), we obtain the following three cases:

5.3.4.1 Case I: $b' \geq 0$

$$I \ = \ R\int_{S_f}^{S_b} \frac{q(c(s), \vec{\nabla}\kappa(s))}{2\sqrt{s-M}}\ exp(-R\int_{S_f}^{s} \frac{\kappa(s')}{2\sqrt{s-M}}ds')ds. \qquad (5.3.13)$$

5.3.4.2 Case II: $b' \in]-2D', 0[$

$$\begin{aligned} I = R\int_{M}^{S_f} \frac{q(c(s), \vec{\nabla}\kappa(s))}{2\sqrt{s-M}}\ exp(-R\int_{s}^{S_f} \frac{\kappa(s')}{2\sqrt{s-M}}ds')ds \\ +R\ exp(-R\int_{M}^{S_f} \frac{\kappa(s')}{2\sqrt{s-M}}ds')[\int_{M}^{S_b} \frac{q(c(s), \nabla\kappa(s))}{2\sqrt{s-M}} \\ exp(-R\int_{M}^{s} \frac{\kappa(s')}{2\sqrt{s-M}}ds')ds]. \end{aligned} \qquad (5.3.14)$$

5.3.4.3 Case III: $b' \leq -2D'$

$$I \ = \ -R\int_{S_f}^{S_b} \frac{q(c(s), \vec{\nabla}\kappa(s))}{2\sqrt{s-M}}\ exp(R\int_{S_f}^{s} \frac{\kappa(s')}{2\sqrt{s-M}}ds')ds. \qquad (5.3.15)$$

Let us first consider the ambient reflection part, we replace on equation (5.3.12), $q(c(h(z)), \vec{\nabla}\kappa(h(z)))$ by $K_a c(h(z)$. We calculate I for each (R, D') and store it into a 2D LUT. For the diffuse reflection part, we replace on equation (5.3.12), $q(c(h(z)), \vec{\nabla}\kappa(h(z)))$ by $K_d c(h(z))\frac{\partial\kappa}{\partial h}h(z)(\vec{N}\cdot\vec{L})$. $\vec{L}$ does not depend on h, thus it can be removed out of the integral I. However, the presence of $\vec{N}$ is problematic, since $\vec{N}$ depends on g. We solve this problem by considering $\vec{N}$ being linear over the integration area (g is quadratic). Therefore, $\exists \vec{A}, \vec{B} \in R^3$ such as:

$$\vec{N} = \vec{A}z + \vec{B} \tag{5.3.16}$$

Thus, the diffuse reflection part of the ray integral is defined by:

$$C_d = \int_0^{D\prime} K_d c(h(z)) \frac{\partial \kappa}{\partial h} h(z)((\vec{A}z + \vec{B}) \cdot \vec{L}) \; exp(-R\tau(0,z))dz \tag{5.3.17}$$

It can be rewritten by:

$$\begin{aligned} C_d &= K_d R(\vec{A} \cdot \vec{L}) \int_0^{D\prime} z c(h(z)) \frac{\partial \kappa}{\partial h}(h(z)) \; exp(-R\tau(0,z))dz \\ &+ K_d R(\vec{B} \cdot \vec{L}) \int_0^{D\prime} c(h(z)) \frac{\partial \kappa}{\partial h}(h(z)) \; exp(-R\tau(0,z))dz. \end{aligned} \tag{5.3.18}$$

Therefore, the integral (5.3.18) can be sampled (up to $\vec{A}$, $\vec{B}$ and $\vec{L}$), in two 2D LUT. $\vec{A}$, $\vec{B}$, and $\vec{L}$ are evaluated, during rendering, in this way:

- $\vec{A} = (\vec{\nabla} b - \vec{\nabla} f)/D'$
- $\vec{B} = \vec{\nabla} f$
- and $\vec{L}$ the light direction.

where $\vec{\nabla} b$ and $\vec{\nabla} f$ are respectively the gradients at S_b and S_f.

Let us now discuss the case for specular reflection. We replace on equation (5.3.12):

$$q(c(h(z)), \nabla\kappa(h)) = K_s c(h(z)) (\frac{\partial \kappa}{\partial h} h(z))^{n_\delta} (\vec{N} \cdot \vec{H})^{n_\delta}$$

where n_δ is the specular exponent. $\vec{H}$ is independent of h, thus it would be interesting if we could separate it from the rest of the integral. However, the presence, here, of the specular exponent n_δ is problematic. We try to overcome this difficulty by applying the following transform:

$$\begin{aligned}(\vec{N}\cdot\vec{H})^{n_\delta} &= ((\vec{B}+\vec{A}z)\cdot\vec{H})^{n_\delta} \\ &= (\vec{B}\cdot\vec{H}+z\vec{A}.\vec{H})^{n_\delta} \\ &= (\vec{B}\cdot\vec{H})^{n_\delta}(1+z\frac{\vec{A}\cdot\vec{H}}{\vec{B}\cdot\vec{H}})^{n_\delta}\end{aligned} \tag{5.3.19}$$

Let us abbreviate $(\vec{B}\cdot\vec{H})^{n_\delta}$ by the scalar u and $(\frac{\vec{A}\cdot\vec{H}}{\vec{B}.\vec{H}})$ by the scalar v. Then we have $(\vec{N}\cdot\vec{H})^{n_\delta} = u(1+zv)^{n_\delta}$ and u is now a multiplicative scalar value that can be shifted out of the rendering integral. Thus, for each sampled value v_j of v, we compute the integral and tabularize it into a 2D LUT. Thus, the specular reflection part of the ray integral can be written as:

$$C_s = K_s R u \int_0^{D\prime} c(h(z))(1+zRv)^{n_\delta}(\frac{\partial\kappa}{\partial h}h(z))^{n_\delta}\ exp(-R\tau(0,z))dz \tag{5.3.20}$$

Therefore, the integral (5.3.20) can be stored (up to $\vec{A},\vec{B}$ and $\vec{H}$) into 2D LUT depending on (u,v).

5.3.5 Realization

In this section, we first show how to generate the opacity LUT, and further the reflection LUT. In the second part, for simplicity, we will consider only the diffuse reflection. The ambient and specular LUT can be implemented in a similar way. Here, we consider the gray values vary quadratically between two sample points, a Simpson rule (order 4) for numerical integration of the subintegral within the sampling interval, is by consequence exact.

5.3.5.1 Implementation of the Opacity LUT

In the previous section, we showed that the opacity α depends, in addition to the gray values at the extremities of a sampling interval, on the parameter R and D'. We also discussed, in the section 5.2.5, how to sample the distance length range. Here, we use the same strategy to sample the range of the parameters R and D', i.e, we use a logarithmic mapping for sampling. Thus, let R_{max} being the maximum possible value for R. We sample the interval $[0, R_{max}]$ on n_r subintervals such as:

$$[0, R_{max}] = \bigcup_{k=0}^{n_r-1} [R_k, R_{k+1}] \tag{5.3.21}$$

And let D'_{max} being the maximum possible value for D'. We sample the interval $[0, D'_{max}]$ on n_d subintervals such as:

$$[0, D'_{max}] = \bigcup_{j=0}^{n_d-1} [D'_j, D'_{j+1}] \tag{5.3.22}$$

Now, let N being the LUT size. Therefore, our algorithm for generating the opacity LUT is the following: For each pairs (R_k, D'_j) and (S_f, S_b), such as $S_b \geq S_f$, we calculate the value of b'. Thus, we know which case, from the 3 ones described previously, has to be considered. Therefore, we evaluate β_i. The evaluation of β_i involves the access of the opacity transfer function lookup table, to get $\kappa(s)$, $s \in [i, i+1]$. In our case, `VGLSampleMaterial` contains mapping parameters from sample (voxel) values to material features like opacity, ambient color, diffuse color, specular color, specular exponent. While the sample (voxel) data type can be chosen freely, the resulting material features are always represented with 32 bit floating point values. The normal range for typical material features like colors is between 0.0 (no intensity) and 1.0 (full intensity). Thus, we lookup the `VGLSampleMaterial` to get respectively

$\kappa(i-1), \kappa(i)$ and $\kappa(i+1)$. Since we assumed the opacity function being piecewise quadratic, we use the de Casteljau's algorithm to get $\kappa(s)$. The pseudo-code of the function implementing the calculation of κ is described in algorithm 11(see Appendix A). Once β_i evaluated for each $i \in [S_f, S_b]$ (e.g in the case I and case II), the results are summed up and stored into the opacity LUT (see pseudo-code of algorithm 9 in Appendix A).

5.3.5.2 Implementation of the Reflection LUT

In this part, we restrict ourselves to the reflection LUT implementation. The approach is similar to the opacity LUT generation. The major difference is the value of β_i. It depends now on the color function c. As we explained for the calculation of the opacity κ, we lookup the `VGLSampleMaterial` to get respectively the colors $c(i-1), c(i)$ and $c(i+1)$. Since we assumed the color function being piecewise quadratic, we use the de Casteljau's algorithm to get $c(s)$. In addition, β_i depends on the exponential term: $exp(-R\int_{inf}^{sup} \frac{\kappa(s')}{2\sqrt{s-M}}ds')$, where the values of inf and sup varies according to the considered case (I or II or III). The exponential term is calculated directly from the already generated opacity 2D LUT, as described in the following equation:

$$exp(-R_k\int_{inf}^{sup} \frac{\kappa(s')}{2\sqrt{s-M}}ds') \quad = \quad 1 - opacity[D_j][R_k][inf][sup]. \qquad (5.3.23)$$

Once β_i evaluated for each $i \in [S_f, S_b]$ (e.g in the case I and case II), the results are summed up and stored into the reflection LUT.

5.3.5.3 Algorithmic Complexity

In our quadratic pre-integration technique for ray casting algorithm, the lookup tables are generated directly from the density values and the parameter R. Let n_R being the number of the considered parameters R. Thus, we calculate the integral for all possible entries in an $n \times n \times n_R$ look-up-table in $O(n^3)$ time.

5.4 Extension to Cubic or Higher-Order Polynomials

The assumption that the scalar values between two sample points along the ray progress quadratically does not match the actual cubic behavior of the scalar values. For this reason, we propose to extend the pre-integration technique for cubic polynomials. The idea is to determine a polynomial of degree three between the entry and the exit point of a sampling interval. We first study the variations of g, if it has a point of inflection, then we split the sampling interval, during rendering process, into two subintervals. Therefore, on each subinterval, we approximate g by a quadratic function such as we can use the already described quadratic model to evaluate the rendering interval result on each subinterval. Finally, we do compositing to get the final result along the whole sampling interval. Notice here that the length of the subintervals is arbitrary and depends on the corresponding cubic polynomial. This problem is solved by the use of different distance length as described before. Since, when applying this approach, there is no limitation on the polynomial degree, we can extend this technique still for higher-order polynomials, proceeding as for cubic polynomials, i.e., we split the actual sampling interval into subintervals where we approximate the corresponding piecewise gray value function by a quadratic model. The partition into subintervals is based on the extrema position of the considered polynomial. Then, for each subinterval, we determine the volume rendering integral, by looking into the already precomputed quadratic 2D LUT. Finally, we composite

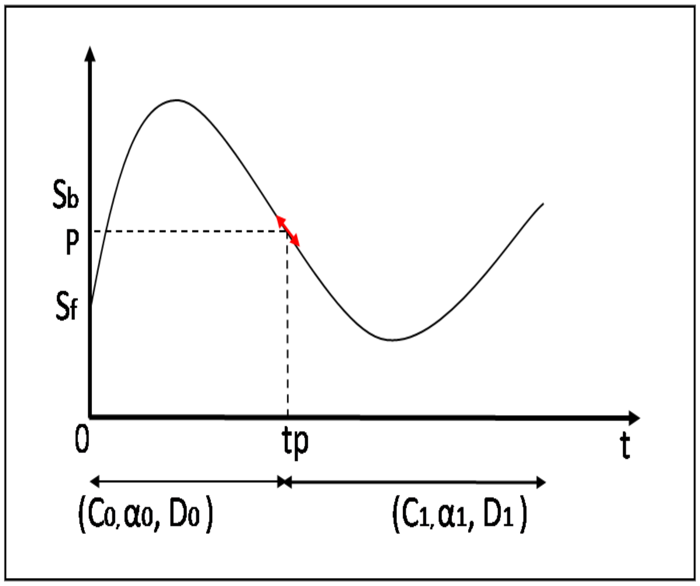


Figure 5.6: Approximation of a cubic curve by a quadratic one

the sub-results to get the final result.

5.5 Shading

We now describe the rendering process for the ray casting algorithm using the pre-integrated technique. First, the user sets the opacity and color transfer function as well as the shading function, in our case the Phong illumination model. Given this information, the pre-integrated lookup tables are generated in a preprocessing phase, as described above. Then, during the rendering process, two different strategies are adopted, based on the degree of the chosen gray value model.

5.5.1 Shading for Linear Gray Value Model

To evaluate shading for the linear model, we first determine, for a given sample interval, its sampling distance as well as its gray values at the boundaries which are obtained by trilinear interpolation. We then use these three values to lookup into the desired pre-integrated lookup table such as the opacity lookup table and one or more among the precalculated reflection lookup tables.

5.5.2 Shading for Quadratic Gray Value Model

In order to evaluate shading for quadratic gray value model, we need to determine, at each sampling interval of distance D, the gray values S_f and S_b at its extremities, as well as the parameter R described previously.

5.5.2.1 Determination of S_f and S_b

Since we assumed the gray value being piecewise quadratic, the use of a quadratic spline interpolation function is an appropriate way, to compute S_f and S_b.

5.5.2.2 Determination of R

We have $R = 1/\sqrt{|a|}$, where a is such as: $g(t) = at^2 + bt + c$. Thus, to determine R, we first, have to evaluate the quadratic polynomial coefficients. For that, let S_m being the gray value at the middle of the sampling interval. The value of S_m is obtained similarly to S_f and S_b, i.e, by quadratic spline interpolation. Then, we consider the following equation system:

$$\begin{cases} g(0) = c = S_f; \\ g(D) = aD^2 + bD + c = S_b; \\ g(D/2) = a\frac{D^2}{4} + b\frac{D}{2} + c = S_m. \end{cases} \tag{5.5.1}$$

By solving this system, we obtain the polynomial coefficients values and deduce the value of R. Then, if $a < \varepsilon$, we read the opacity and reflection values of the current sampling interval from the lookup tables generated for g being linear. Otherwise, we lookup into the opacity LUT generated for quadratic model, to get the opacity value inside the sampling interval and into the reflection LUT to get the color value for ambient, diffuse or specular reflection. Finally, we sum up the different color values to obtain the Phong shading result.

5.6 Results

The proposed pre-integrated volume rendering algorithm was implemented on a computer which runs under Windows XP on a 1700MHz Intel Pentium(M) with 1.5 GB of RAM and which has an ATI MOBILITY FIRE GL T2 graphics card, using C++ and VGL (www.volumegraphics.com) as underlying library. For our experiment, we choose 8 bit data sets: the Bonsai (CT) of size ($256 \times 256 \times 256$ voxels), the Engine (CT) of size ($256 \times 256 \times 128$ voxels) referred by Engine 1; and the Engine (CT) of size ($128 \times 192 \times 256$ voxels) referred by Engine 2.

5.6.1 Algorithmic Performance

5.6.1.1 Linear Look-up-table Generation

For the linear gray value model, the proposed method for pre-integrated volume rendering uses three-dimensional lookup tables to store ambient, diffuse, specular reflection and opacity. The entries of each lookup table depend on three values: the entry and exit density values and the length between them. Since the lookup tables have to be recomputed whenever the transfer function changes, the interactivity is a relevant factor in the pre-integration algorithm. As shown in the table 5.1, in comparison to

Gray model	n	Preprocessing time (s)
Linear	32	0.01
	64	0.05
	128	0.2
	256	0.791

Table 5.1: Relationship between LUT computation time and linear pre-integrated lookup table dimension n.

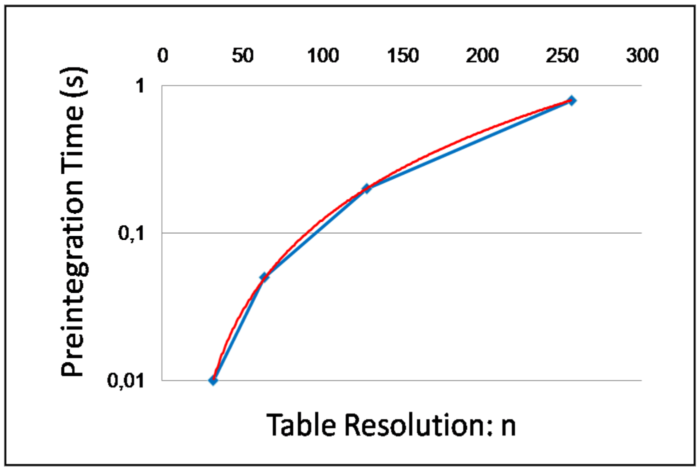


Figure 5.7: Relation between the preintegration time and the LUT size (Blue curve) and a fit of a quadratic function to it (Red curve).

standard numerical pre-integration [9], our proposed pre-integration precomputation achieves a speed up of approximately 10 times, without use of hardware acceleration. Figure 5.7 shows that the preprocessing time is a quadratic function of the lookup table resolution, which validate the computed complexity of our linear preintegrated approach, namely $O(n^2)$. The smaller the dimension n, the faster is the pre-integration algorithm.

Gray model	n	Preprocessing time (s)
Quadratic	32	7.611
	64	30.013
	128	119.953
	256	496.594

Table 5.2: Comparing computation time in second for $n \times n$ quadratic pre-integrated lookup table.

5.6.1.2 Quadratic Look-up-table Generation

For the quadratic gray value model, our proposed method for pre-integrated volume rendering uses three-dimensional lookup table to store ambient, diffuse and opacity and four-dimensional lookup table to store specular reflection. In addition to the entry and exit density values and the sampling interval distance, the three-dimensional lookup tables depend also on R defined in section 5.3. The specular lookup table depends in addition to those three parameters, on v as demonstrated in section 5.3.4. In our case, the rendering is performed using a ray-casting algorithm. Thus, a constant sampling interval length is used. The large dimension of the specular lookup table, which results in high computation time, prevents our algorithm from achieving good performance. Thus, we will not consider, in the following, the computation time due to the specular part. For the rest of the tables, when a change on the reflection coefficient (ambient,diffuse) occurs, only an update of the concerned lookup table is necessary. Thus, the updating time is minimized. Even those simplifications, comparing to the linear model, the preprocessing time for the quadratic gray value model is still higher as shown in table 5.2. Figure 5.8 shows that the preprocessing time is a cubic function of the lookup table resolution, which validate the computed complexity of our quadratic preintegrated approach, namely $O(n^3)$.

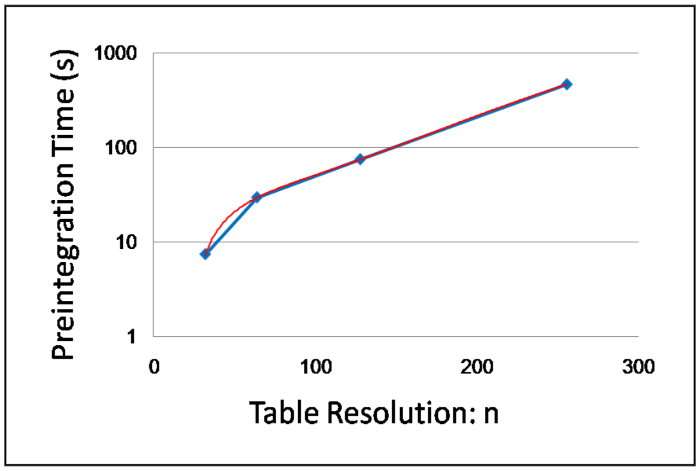


Figure 5.8: Relation between the preintegration time and the LUT size (Blue curve) and a fit of a cubic function to it (Red curve).

5.6.1.3 Influence of Distance Parameter

For the third parameter of pre-integrated lookup tables, i.e., the sampling interval distance D, the number of the distances to be considered depends on the volume rendering algorithm. For example, for the ray casting algorithm, the sampling distances along the ray are constant, thus one need only a 2D pre-integrated lookup table with the third parameter being constant. For the projected tetrahedra algorithm, $D \in [0..\sqrt{3}]$, therefore one has to consider different sampling distances as we discussed earlier in this chapter. In figure 5.9, we show how the number of the considered distances influences the pre-integration time for a 256^2 pre-integrated lookup table.

5.6.1.4 Influence of the Parameter R

Let us now discuss the range of the parameter $R = 1/\sqrt{\mid a \mid}$. Our experiments showed that $|a| \in [0..a_{max}]$ with $a_{max} = 55.23$. In section 5.5.2, we assumed that under a

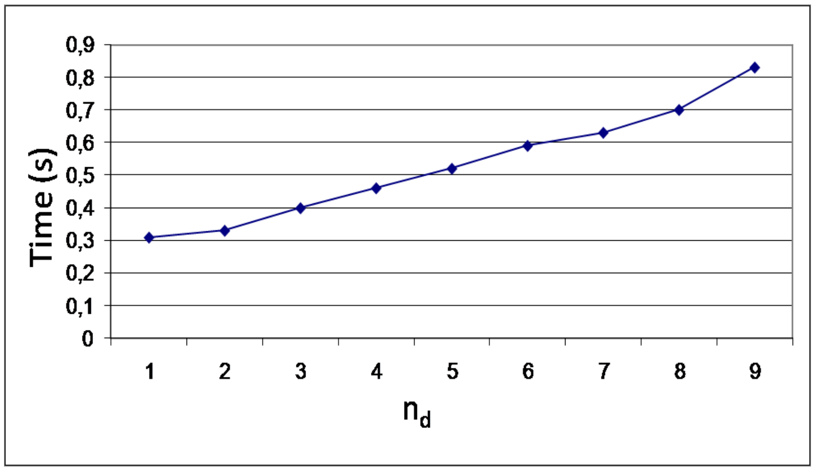


Figure 5.9: Relationship between the LUT size in distance parameter direction and the linear LUT pre-integration time

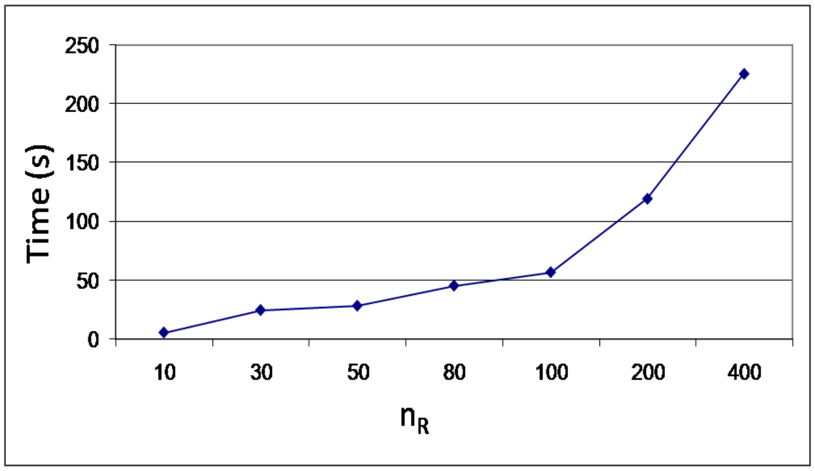


Figure 5.10: Influence of the R parameter number n_R on the pre-integration time en seconds

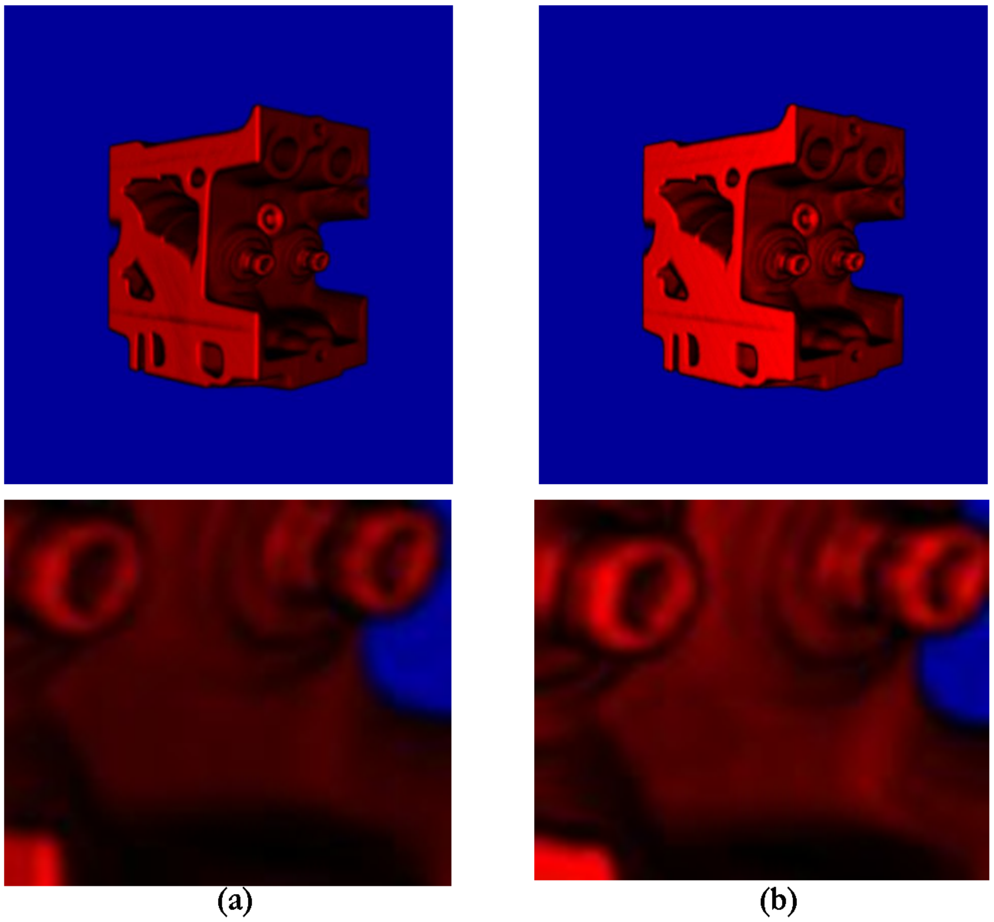


Figure 5.11: Image quality: (a) represents an image rendered with logarithmic subdivision of R. (b) represents an image rendered with $n_R = 400$

Subdivision approach		Time (s)
Sampling using n_R steps	10	5.698
	30	24.726
	50	28.441
	80	45.516
	100	56.942
	200	119.381
	400	225.594
Logarithmic		4.076

Table 5.3: Comparison between two different approaches for the subdivision of the R range and their influences on the lookup table generation speed

certain threshold value a_{th} of a, the density function g is assumed linear. Thus, if $a \in [0, a_{th}]$, there is no need to lookup into the quadratic pre-integrated LUT, to get the corresponding opacity and reflection values. Otherwise, if $a \in [a_{th}, a_{max}]$, i.e, $R \in [0.134..7.76]$, we can suppose that $R \in [R_{min}, R_{max}]$ with $R_{min} = 2^{-3}$ and $R_{max} = 2^3$. Thus, the quadratic pre-integrated lookup tables have to cover the whole range $[R_{min}..R_{max}]$. For that, we consider a logarithmic subdivision of the parameter R, as we did for the interval length. Another alternative is to subdivide the R range in equal subintervals. We tested both approaches and as shown in figure 5.10 and table 5.3, we found out that the logarithmic subdivision gives a faster lookup tables generation than the second approach and this for a better image quality (see figure 5.11).

Gray model	n	Image Error (%)	Image No
Linear	32	0.1624	5.13.a
	64	0.108	5.13.b
	128	0.0678	5.13.c
	256 (Image reference)		5.13.d

Table 5.4: Integration accuracy of our linear pre-integration approach: the image error is defined by the average error metric

5.6.2 Image Quality

To numerically evaluate the image quality as function of the lookup table size, we use the average error metric as used by Danskin and Hanrahan [8], given by:

$$Err_{im} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \mid I_r(i,j) - I(i,j) \mid}{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} I_r(i,j)} * 100\% \tag{5.6.1}$$

where $I_r(i,j)$ is a pixel value in the reference image which in our case corresponds to an image rendered with lookup table of size 256×256; $I(i,j)$ is a pixel value in the image rendered with lower lookup table resolution; N and M are image dimensions.

5.6.2.1 Linear Look-up-table Generation

Table 5.4 shows that the image quality of our algorithm is affected by the lookup table resolution used during the pre-integration algorithm. Figure 5.12 shows that image error increases with smaller table.

During rendering phase, whenever one or both gray values at the extremities of a sample interval $[t_k, t_{k+1}]$ fall between two entries of the pre-integrated LUT, a bilinear interpolation between four adjacent values of the table is necessary to get the approximated opacity or reflection value for the interval $[t_k, t_{k+1}]$. Hence, we deduce that large table sizes are desirable because the larger the table, the smaller

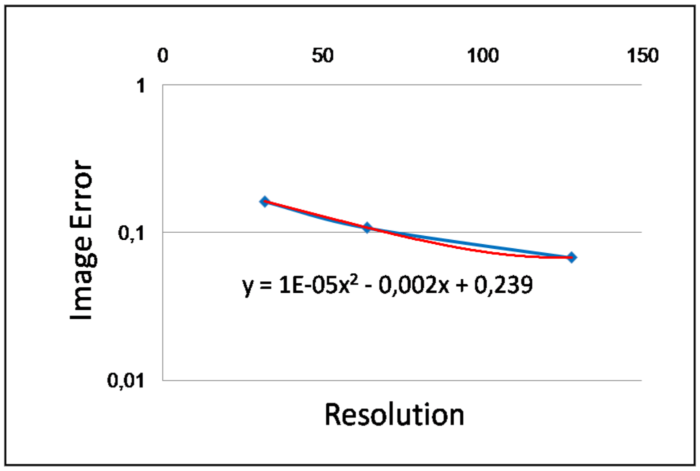


Figure 5.12: Relation between image error and the LUT resolution (Blue curve), for linear gray value model and a fit of a quadratic function to it (Red curve).

Gray model	n	Image Error (%)	Image No
Quadratic	32	0.68	5.14.a
	64	0.27	5.14.b
	128	0.0844	5.14.c
	256 (Image reference)		5.14.d

Table 5.5: Integration accuracy of our quadratic pre-integration approach: the image error is defined by the average error metric.

such interpolation error will be. By consequence, a higher resolution lookup table results on a better image quality.

5.6.2.2 Quadratic Look-up-table Generation

To achieve better results, we can decrease the lookup tables resolution. The relation between the image quality and the LUT size is shown in the table 5.5. As for the linear gray value model, the figure 5.15 explain that the image error decreases whenever the LUT resolution increases.

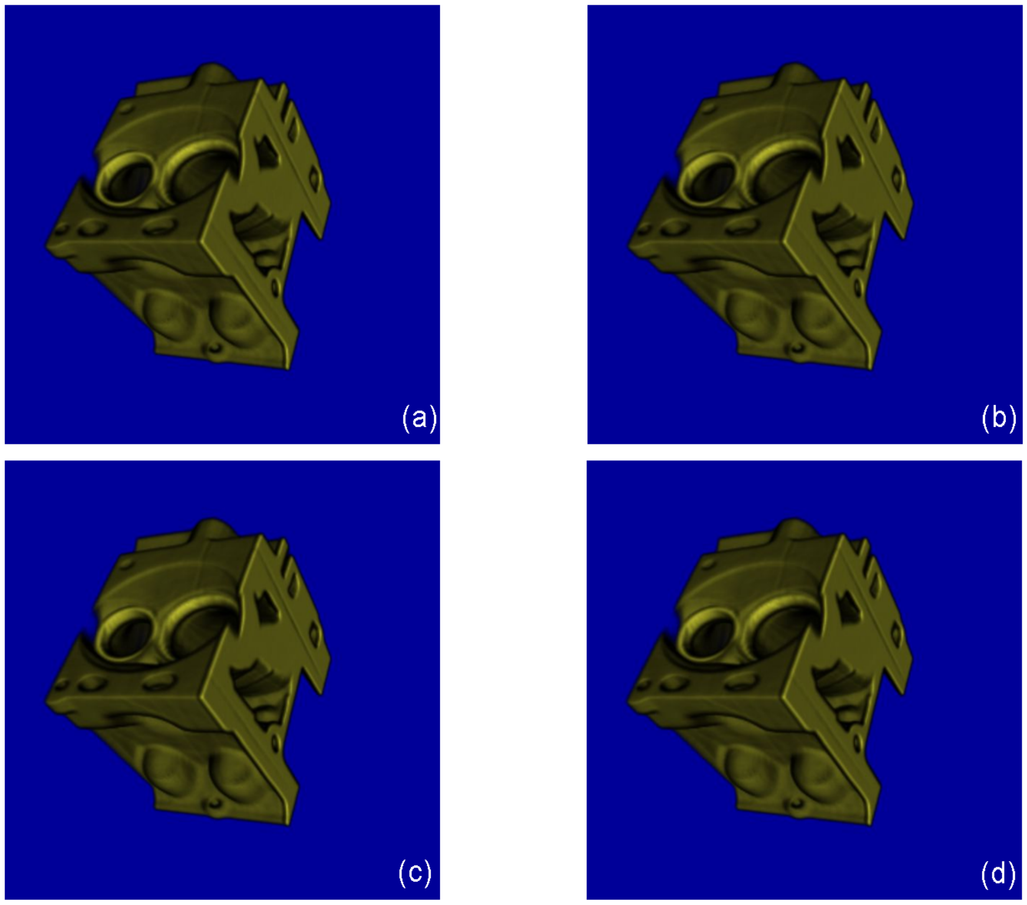


Figure 5.13: Images rendered using different resolutions for linearly pre-integrated lookup tables:Image *a* corresponds to resolution 32.Image *b* corresponds to resolution 64.Image *c* corresponds to resolution 128.Image *d* corresponds to resolution 256

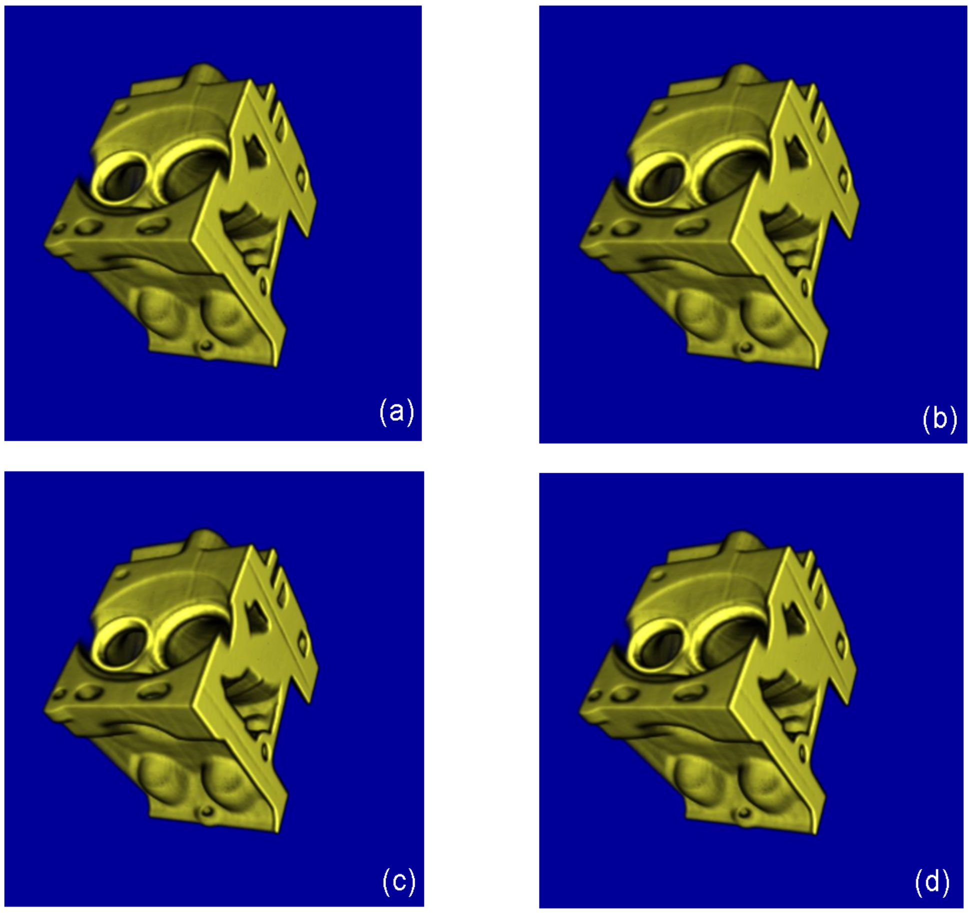


Figure 5.14: Images rendered using different resolutions for quadratically pre-integrated lookup tables:Image *a* corresponds to resolution 32.Image *b* corresponds to resolution 64.Image *c* corresponds to resolution 128.Image *d* corresponds to resolution 256

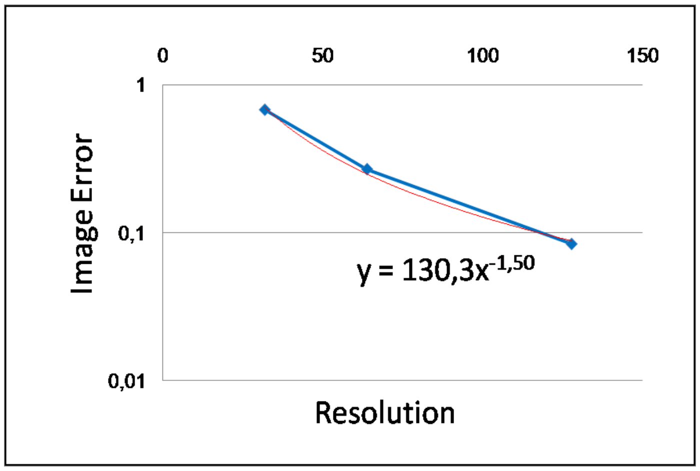


Figure 5.15: Relation between image error and the LUT resolution (Blue curve), for quadratic gray value model and a fit of a quadratic function to it (Red curve).

5.6.3 Discussion and Analysis: Optimization of the LUT size

Earlier in this section, we described the influence of the LUT size on our algorithmic performance and image quality. In this part, we generate a model to find out the optimal LUT size given a volume data set.

In our experiment, we considered three opaque and semi-transparent data sets. Our goal is to set a relationship between data sets parameters and the optimal LUT size for preintegration. For our analysis, we consider two properties, namely the size and the transparency of the data sets. In the following, we will first discuss the size parameter, then the transparency one. Let us consider the three data sets of different sizes and the same transparency property, i.e, opaque or semi-transparent. Figure 5.16 shows that, for an image error threshold equal to 0.5%, bigger data sets converge faster to the optimal LUT size, in case of semi-transparent objects, while the opposite

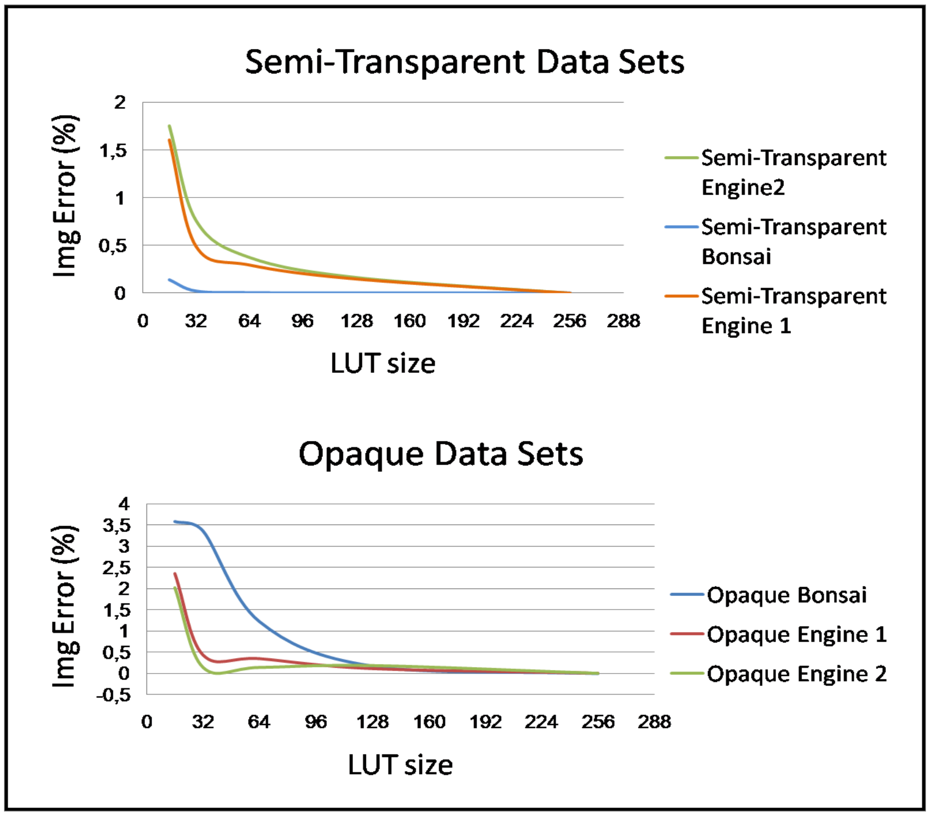


Figure 5.16: Influence of the volume size on the resulting image quality for both opaque (below) and semi-transparent (above) data sets.

situation occurs in case of opaque objects, i.e, smaller data sets converge faster than bigger ones. Let us now vary the transparency of the data sets, while keeping the same size. According to figure 5.17, we notice that the semi-transparent bonsai needs less sampling distance (32) than the opaque one (96). For engine data sets, both semi-transparent and opaque engine converge to the same optimal LUT size (32). We now interest in the LUT size as function of both the size and the transparency parameters. For this purpose, we consider as size parameter the 3D data set size in

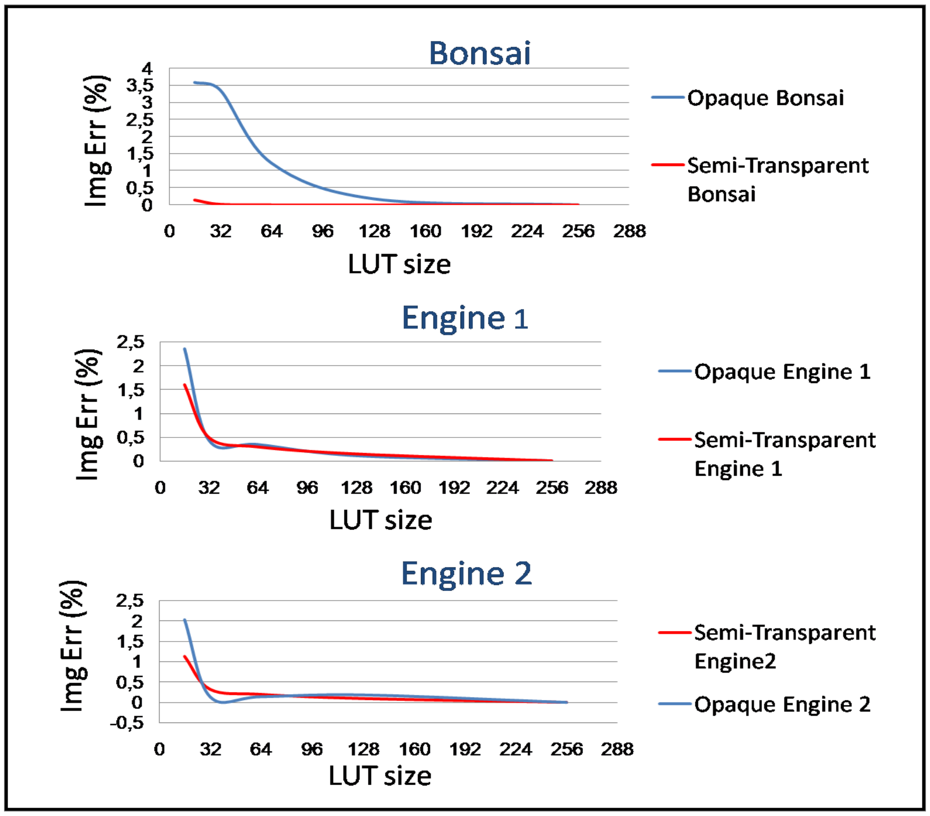


Figure 5.17: Influence of the volume transparency on the resulting image quality for different data sets.

LUT size	ST Teapot	Opaque Engine 1	Opaque Engine 2	ST Engine2	Opaque Bonsai	ST Bonsai
16	2,3605	2,0314	1,1328	3,5821	0,1364	1,6031
32	0,4312	0,1396	0,3224	3,358	0,02	0,4897
64	0,3461	0,1394	0,1973	1,2248	0,0035	0,2945
128	0,1105	0,1838	0,0951	0,1775	0,00062474	0,1479

Table 5.6: Influence of the LUT dimensions on the Image error.

bytes and as transparency parameter the mean of the opacity function.

$$m = \frac{1}{b-a} \times \int_a^b f(x)dx \tag{5.6.2}$$

By using the different data sets we listed above, we obtain from the values in the table 5.6, the function $f(x, y) = z$ with x being the object size, y the transparency parameter and z the corresponding LUT size. Figure 5.18 shows the behavior of this function. To test the obtained function, we consider a semi-transparent teapot (artificial CT) of size ($256 \times 256 \times 256$ voxels) and by numerical integration, we calculate the corresponding optimal LUT size for this data on MATLAB.

$$Z = interp2(x, y, z, 16777216, 0.1404)$$

The obtained result is 128 which fit well with the result obtained in the figure 5.19. Thus, we have derived a function which depends on two data set parameters and returns the optimal pre-integrated LUT size. Thus, one can accelerate the LUT generation time by choosing the adequate LUT resolution without affecting the image quality.

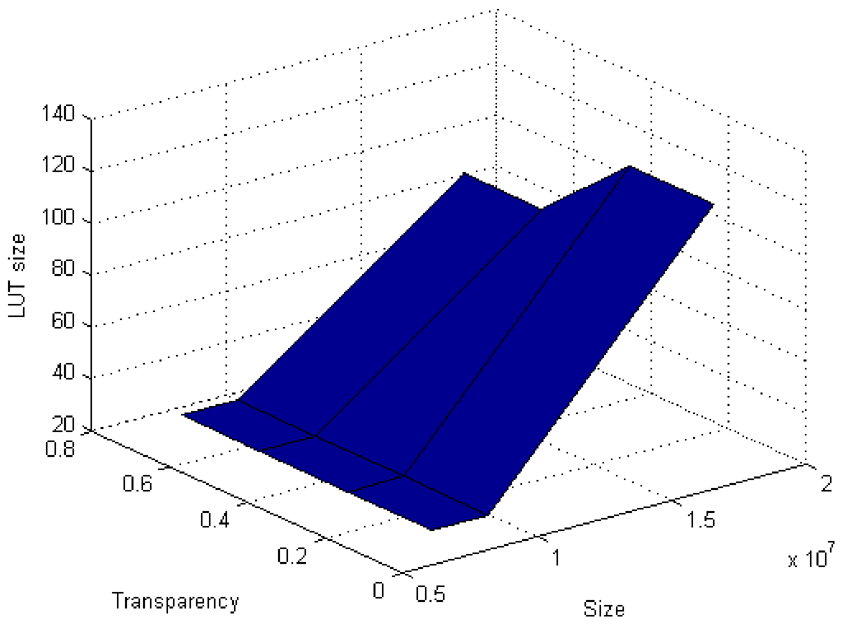


Figure 5.18: Influence of volume transparency and size parameters on pre-integrated LUT dimension.

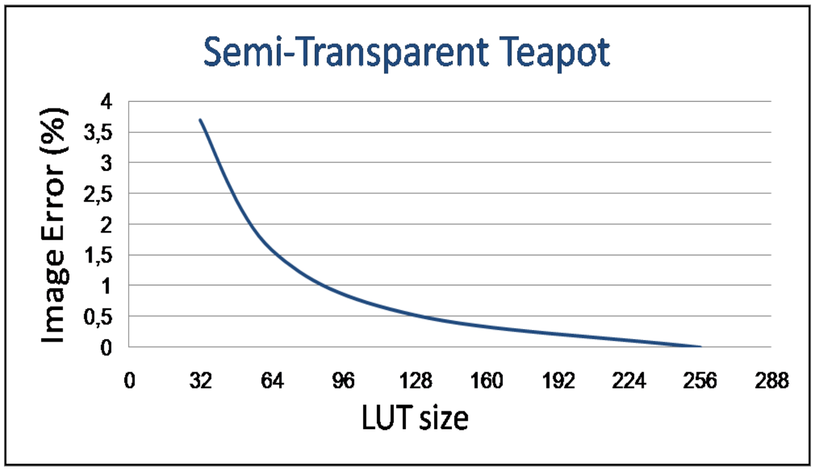


Figure 5.19: Influence of the LUT size on the image error for a semi-transparent teapot (256 × 256 × 256 voxels) data set.

5.7 Conclusion

In this part, it has been shown that the Phong shading can be integrated into the pre-integrated volume rendering frame without any problems and without having any approximations concerning the normal directions depending on the opacity and the absorption being included in the formula. The shading result for any interval can be obtained by dedicated compositing of pre-computed values read from the 2D LUTs. Due to the pre-computation, shading calculations remain cheap and, therefore, they can be applied to arbitrary complex scenes. As long as the shading can be applied, and thus even more complex shading models are applicable without any performance degradation during visualization. We also derived a model to determine the optimal LUT size given a volume dataset.

Chapter 6

Coherence Encoding

6.1 Introduction

Fast volume rendering is today one of the major challenges of visualization. Because of the large size of volume data, rendering is slow. However, there is generally a high degree of similarity or spatial coherence in the data. In fact, in many data sets, properties do not change drastically but rather in a smooth or continuous way. Thus, exploiting coherence is one of the most efficient techniques to accelerate volume rendering. Coherence is based on the principe of locality, whereby parts do have the same or similar characteristics. Different types of coherence exist. [21] gives a good survey of them. In this chapter, we limit ourselves to spatial coherence. Spatial techniques seek to explore coherence within the data to accelerate rendering. In volume data, they describe spatial homogeneity. These are a consequence of constant or slow varying relationships in the spatial arrangement of data. For example, if one consider the opacity property of the data, the coherence can be described by a constant relationship, i.e, voxels, in a certain range, have the same opacity value, e.g, they are either empty (emptiness) or homogeneous (homogeniety). Coherence can also be a

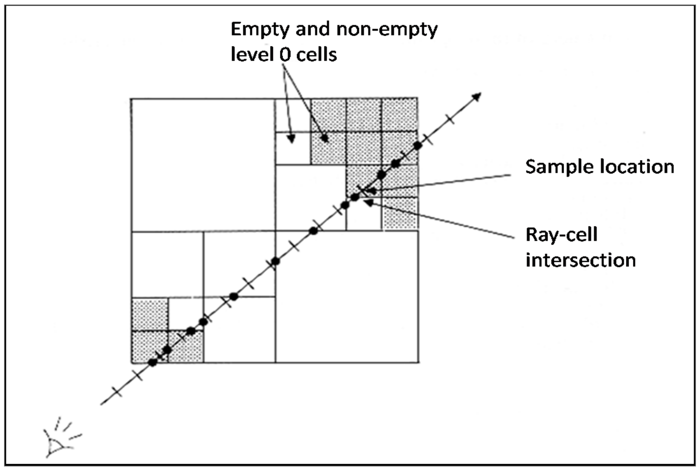


Figure 6.1: Ray tracing of hierarchical enumeration

slow varying relationship that can be expressed by linearity, i.e, it exists linear transition between opacity values of voxels in small neighborhood. Homogeneity in volume data have been exploited in a variety of different methods and techniques. Data can be stored more efficiently by eliminating redundancies thanks to coherence. In the following, techniques and data structures for exploiting coherence in volume rendering are described. Some specific data structures have been developed that are well suited to exploit coherence properties. The most popular spatial subdivision technique is that of the octrees [17, 51]. It describes an adaptive hierarchical subdivision of 3D space. An octree can adjust the volume to varying levels of coherence by subdividing incoherent regions further. This method was applied to ray casting volume rendering algorithm by Levoy [36](see figure 6.1). In preprocessing, he uses an octree to describe the empty regions in a volume. Then, during rendering, the ray is traced through the octree using information within the nodes to skip over the empty voxels. Further, Subramanian [55] extended this technique to render efficiently volumes where the data of interest is distributed sparsely. He suggested that the computationally expensive traversal of the ray through the octree can be avoided by storing the octree information at the empty voxels as uniformity information. In that way, an additional value is stored into the empty voxels to indicate in which level of the octree they are. This value is then used to cause the ray to skip to the next first voxel beyond the uniform region. Subdivision techniques proceed top-down, subdividing a given space into smaller subspaces. Those techniques produce good (non overlapping) hierarchies but weak bounds. To overcome these disadvantages, further techniques have been developed to encode homogeneity in volume data. Methods were discussed by [63] in which additional data or the volume itself is used to store proximity flags or values which present distances which will enable an accurate ray traversal algorithm. The distance values indicate how far a ray can jump without encountering an object. These methods suffer from the fact that 3D preprocessing is required whenever a change on the data occurs. Where most of the previous techniques were restricted to ray casting, Lacroute [34] proposed a fast shear warp algorithm where in the preprocessing step, the volume is encoded by run-lengths in voxel scanlines which is used in the compositing step to skip empty voxels. His method made it possible to use coherence in both voxel scanlines to implement space leaping and in the image to implement early ray termination. All the approaches we have mentioned until now exploit only one of the three aspects of coherency namely the emptiness. Further, Freund et al [14] proposed a method which allows to encode both emptiness and homogeneity on the voxels. However, their method needs to perform segmentation in preprocessing step to determine homogeneity in the volume, which is a very time consuming process. Recently, Chen [5] proposed a method to exploit linear coherence in volumes, in both shear warp and ray casting algorithm. In his new shear warp algorithm, he used a simple ambient shading model and an intensity-interpolation scheme (Gouraud shading). By consequence the images lack sometimes of liveliness. As for the performance, it has been proved that although the preprocessing time is faster, the rendering process is almost the same as in Volpack [34]. This chapter describes the developments made in both shear warp and ray casting algorithm as well as their implementation on the platform independent computer graphics library VGL.

6.2 Accelerating Shear Warp

6.2.1 Implementation of earlier shear warp algorithm for parallel projection

In this section, we give an overview about the basic ideas of the original shear warp algorithm [34], which we later develop in our new implementation. To exploit coherence on the shear warp algorithm, Lacroute [34] uses a run-length-encoding (see figure 6.2) to encode the volume data. The run-length-encoding is composed of three data structures:

1. a **run-length array** to store the lengths of the empty and non-empty runs of voxels within a voxel scanline. Opacity values are continuous from 0.0 to 1.0. A value of 0.0 indicates a completely transparent, i.e, empty voxel, a value of 1.0 indicates a completely opaque voxel.

2. a **voxel array** where is stored the non empty voxels one by one. The empty

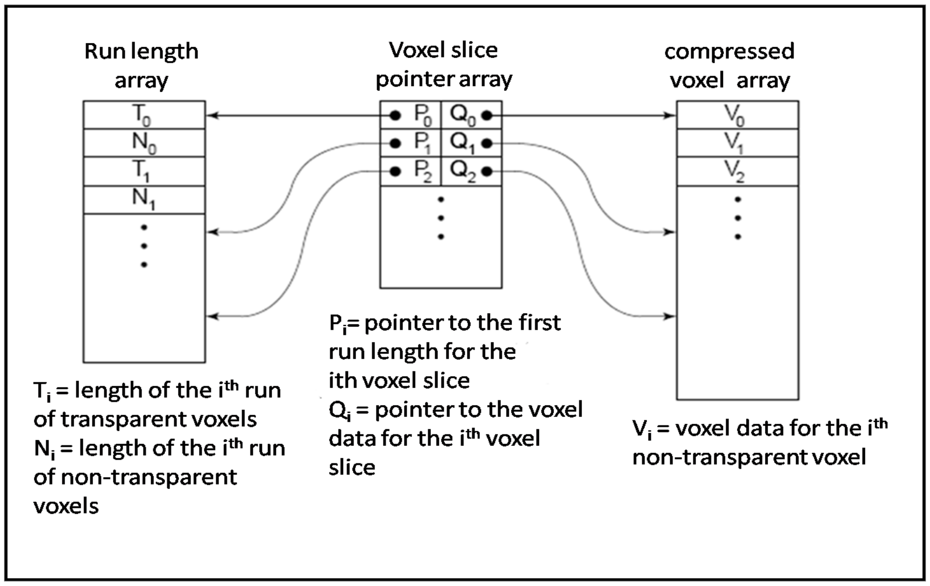

Run length array
Voxel slice pointer array
compressed voxel array
T0
N0
T1
N1
P0 Q0
P1 Q1
P2 Q2
V0
V1
V2
Pi= pointer to the first run length for the ith voxel slice
Qi = pointer to the voxel data for the ith voxel slice
Ti = length of the ith run of transparent voxels
Ni = length of the ith run of non-transparent voxels
Vi = voxel data for the ith non-transparent voxel

Figure 6.2: The 3 data structures of the run-length-encoded volume

voxels are not saved because they do not participate in the rendering process.

3. and finally for each slice, a **voxel slice pointer array** with two entries, each entry contains a pointer: the first one points to the first run-length for the slice in the run-length array, and the second one points to the first non-empty voxel of the slice in the voxel array.

Three precomputed run-length-encodings are used for each major axis if the viewing direction changes considerable. Then, to take advantage of coherence in the image space, run-length-encoding of the intermediate image is used. This data structure is constructed dynamically during rendering. The run-length-encoding consists of an offset stored with each opaque pixel, i.e., each pixel whose opacity exceeds a user specified threshold. The offset points to the next non-opaque pixel in the same scanline, as shown in figure 6.3. These offsets are used during rendering to know how

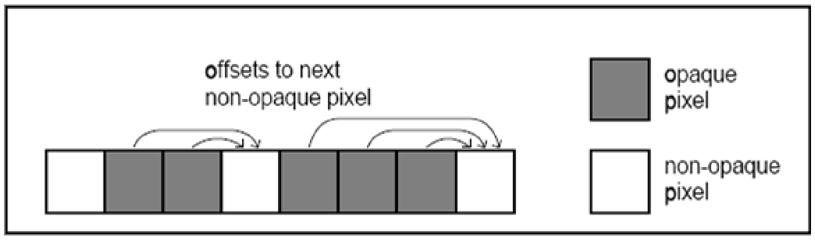


Figure 6.3: Offsets stored with opaque pixels in the intermediate image scanline

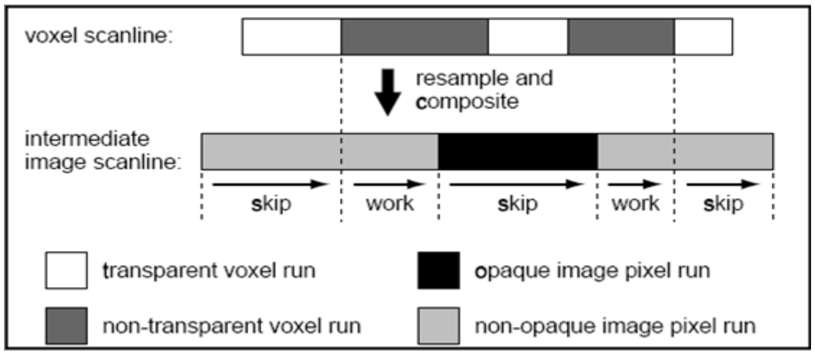


Figure 6.4: Traversal of voxel and image scanlines: resampling and compositing only performed when voxels in run are non-empty and pixels in the intermediate image are non-opaque

much pixels have to be skipped in the current image scanline. By using both data structures above and since the voxel scanlines in the sheared volume are aligned with the pixel scanlines in the intermediate image, the rendering process is implemented as given by the figure 6.4. For each slice of the volume, the algorithm marches simultaneously through the voxels and the intermediate image, in scanline order. The run-length-array of the voxel data is used to skip transparent voxels, while the run-length-encoding of the intermediate image is utilized to skip opaque pixels, using the offsets stored with the pixels. Only the non-transparent and visible voxels are shaded, resampled and composited.

6.2.2 Accelerating Shear Warp by Coherence Encoding

In the basic shear warp algorithm, in the rendering phase, the empty runs of voxels are skipped (space-leaping), while the voxels in the non-empty runs are processed one by one. Even this acceleration method works well for volumes where the voxels are classified so that they are mostly opaque or transparent, it is not the case for volumes with many semi-transparent regions. To keep a high rendering speed for all volume classification functions, Chen [5] exploits two types of coherence in the voxel data set, namely the homogeneity [14] and the linearity. In fact, some voxels in the volume scanlines can have approximately the same opacity. The idea is to linearize the opacity curve along the voxel scanline as shown in figure 6.5. First, a maximal allowed error is defined. Then, given this threshold of error, a distance value d for each voxel is determined, within which the opacity curve can be approximated by a linear function. For this purpose, we first implemented a function which determines the linearization error Err_{lin} between 2 voxels a and $a+d$, given by:

$$Err_{lin}(a, a+d) = \sum_{i=1}^{d-1} |opacity(a+i) - (opacity(a) + \frac{i}{d}(opacity(a+d) - opacity(a)))|$$

The first modification is the additional coherence distance data stored in the voxel. The second is that the number of the voxels saved in the voxel data array will decrease. In fact, we only have to save the boundary voxels of each linearized segment. The voxels in between do not need to be saved, since we can obtain the required information there for shading by a linear interpolation function that we have implemented. This modification in voxel array data leads in consequence to a modification in the voxel slice pointer array, precisely the pointer to the first voxel data in the voxel array.

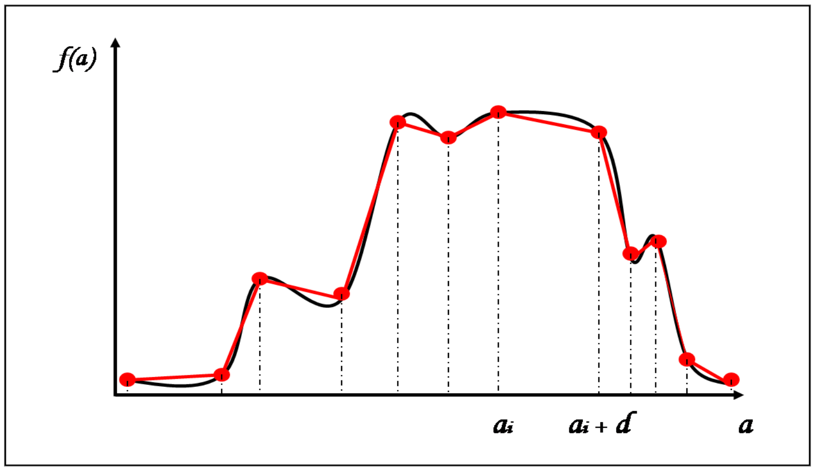


Figure 6.5: Linearization of the opacity curve along a voxel scanline: the x axis represents the gray values along a voxel scanline and $f(x)$ represents the opacity function. The black curve is the continuous opacity transfer function. The red curve defines the piecewise opacity function acquired by connecting the sample values (red dots) obtained by linearization of the continuous function

The run length array is kept unchanged. Those modifications being accomplished, we obtain the new run length encoding with coherence. Thereafter, we used this new run length encoding volume in the render process. We first experimented with this implementation of the shear warp algorithm where the intermediate image scanline is sequentially traversed with one voxel scanline and the second time with two voxel scanlines at the same time.

6.2.3 Implementation

Comparing to the original implementation of the shear warp algorithm [34], some modifications were introduced by Chen [5]. On one hand, he saved memory to store the non-empty voxels: instead of storing the precomputed normals, the opacity and gray value for each voxel, he only stored the opacity value and the coherence distance for each voxel in the voxel data array. The number of voxels inside the voxel data array is also minimized since only voxels on the boundary of the linearized segment are stored. The colors and opacities of the voxels in between are obtained by linear interpolation. On the other hand, to accelerate rendering process, he adopted an ambient light whose reflection is proportional to the voxel opacity as shading function. This simple light model was previously adapted on some existing rendering algorithm [39]. Then, an intensity interpolation(Gouraud shading) is used to evaluate color for each voxel. During the rendering process, this algorithm marches only through one voxel scanline and the intermediate image scanline. In our implementation, we adopt the coherence encoding considering two voxel scanlines in parallel, that means two input scanlines are traversed and decoded simultaneously. Bilinear interpolation between the two scanlines are used to evaluate the final color of the voxel to be composited to the intermediate image. We first encode the coherence in a voxel scanline, as shown in algorithm 3.

After encoding linearization, the voxel data stored in the voxel array changes as shown in figure 6.6. We only save the boundary voxels of each coherent region, with their calculated coherence distance. Thus, the number of saved voxels decreases. During rendering process, two adjacent voxel scanlines are processed in parallel. When a run of empty voxels is present in one of them, corresponding voxels in the other voxel scanline as well as pixels in the intermediate image scanline are simply skipped. Voxels in adjacent scanlines are also skipped whenever the corresponding pixels in the intermediate image scanline are already opaque. Otherwise,

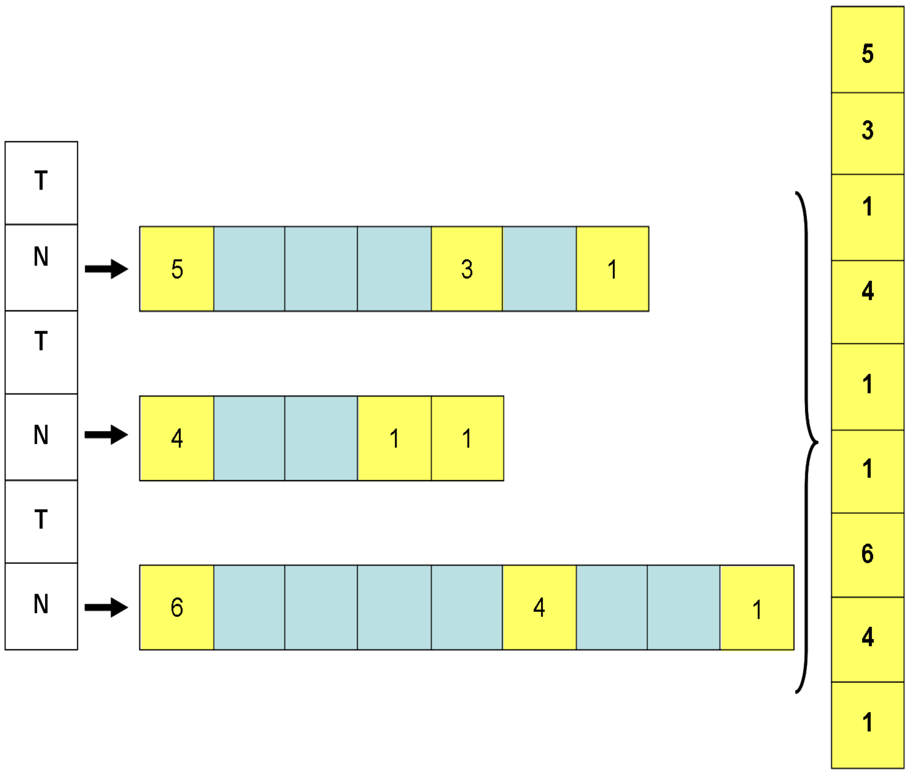


Figure 6.6: The voxel array after coding distance: only the boundary voxels (yellow) of the coherent region (green) are saved on the voxel array

Algorithm 3 Calculate Coherence($*rled, l$): Pseudo code for encoding coherency in voxel scanline: First, the algorithm calculates the distance where the error is still below a threshold. Then, it saves for each voxel in between the two boundaries of the linearized scanline part, its coherence distance

```
for $i = 0$ to $l$ do
  $d = 0;$
  while $(CalculateCoherence(rled, d) \leq Err_{lin}$ $and$ $i < l)$ do
    $i++;$
    $d++;$
  end while
  $d--;$
  for $j = d$ to $0$ do
    $distance(rled) = j;$
    $rled+ = Byte\_per\_Voxel$
  end for
end for
```

when marching through non-empty runs of the two adjacent voxel scanlines, we consider a run of voxels of length corresponding to the minimum of both run lengths. Then, for each voxel present in one of both voxel scanlines, we check if it is saved or not on the voxel array. If it is the case, the corresponding color and opacity of the voxel are retrieved. Otherwise, the voxel is in the coherent region. Thus, a linear interpolation between the two saved boundary voxels in the corresponding scanline, is done to get the adequate color and opacity values. Thus, given four voxel data and using the already calculated weights, which are constant because of the parallel projection, we evaluate by bilinear interpolation the voxel data to be composited to the corresponding pixel in the intermediate image scanline, as shown in figure 6.7.

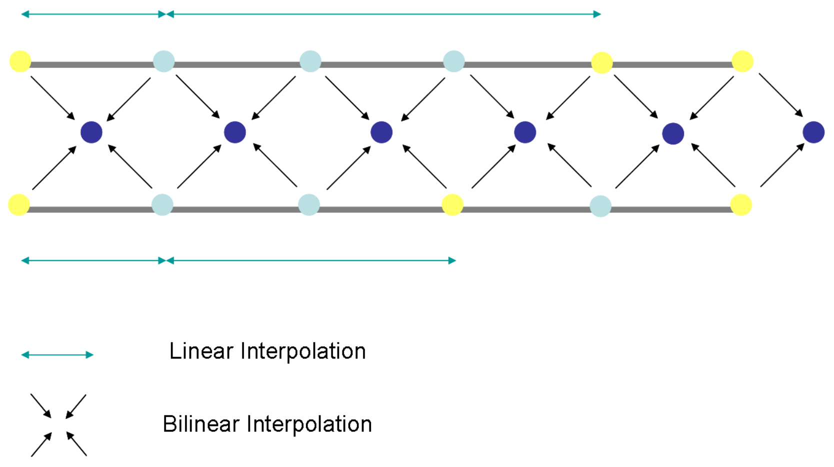


Figure 6.7: Traversal of two adjacent voxel scanlines

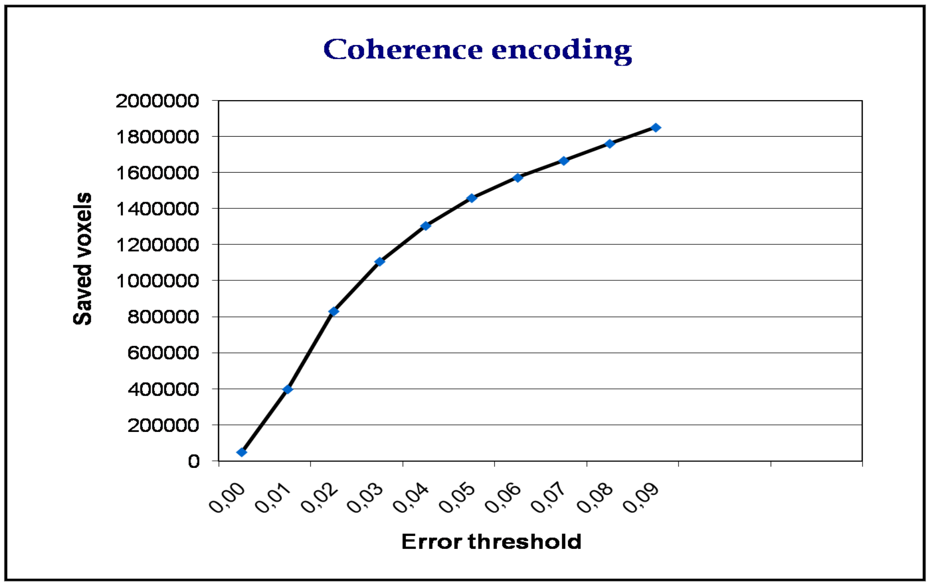


Figure 6.8: Number of saved data

6.2.4 Results

The statistics made on the engine data set with 2 classifications types (semi-transparent and opaque) lead to the following results:

1. Less memory space used to store voxel data: (see figure 6.8).

2. To numerically evaluate the image quality, we used metrics in terms of image error. We simply used the root mean square (RMS) to determine this error and as result we notice that there are no relevant changes in image quality for a small error threshold as shown in figure 6.9.

The influence of the coherence error on the image quality is reflected by the curve in figure 6.9. We notice that, while increasing the coherence error, the image error remains the same. This is due to the fact that the coherence error and the coherence

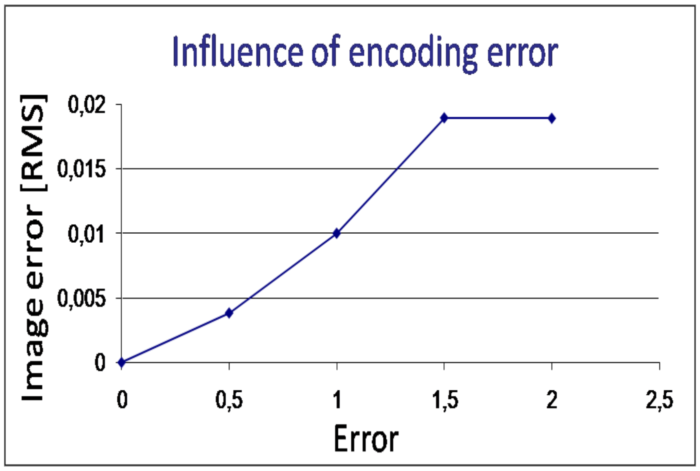


Figure 6.9: The influence of encoding error

encoded distance are proportional. Thus, raising the coherence error implies raising the encoded distance. However, this distance can not increase indefinitely due to the presence of the empty runs in the voxels scanlines. Therefore, the coherence encoded distance augments whenever the coherence error augments, until a new run of empty voxels starts. Thus, on a certain stage, increasing the coherence error has no more effect on the image error. Finally, this explains why the curve in fig 6.9 becomes constant for high coherence error values.

The image quality of our algorithm is affected by the error threshold value used during encoding the coherence in the shear warp algorithm. Figure 6.10 shows the difference between images rendered with different coherence encoding errors for the opaque engine data set, while the figure 6.11 shows the same but for the semi-transparent engine data set.

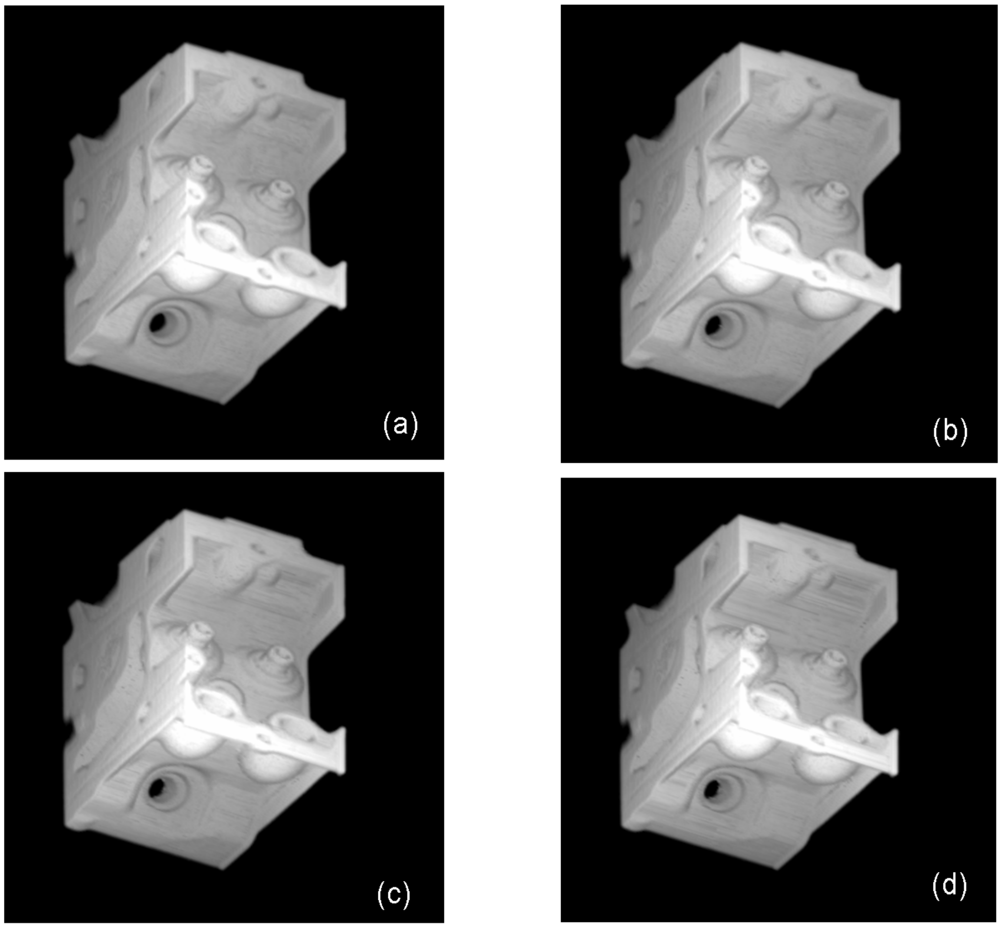


Figure 6.10: The influence of encoding error.Image a is the reference image rendered without coherence encoding.Image b,c,d are images rendered with coherence encoding using an error threshold value respectively equal to $0.2, 0.5$ and 1

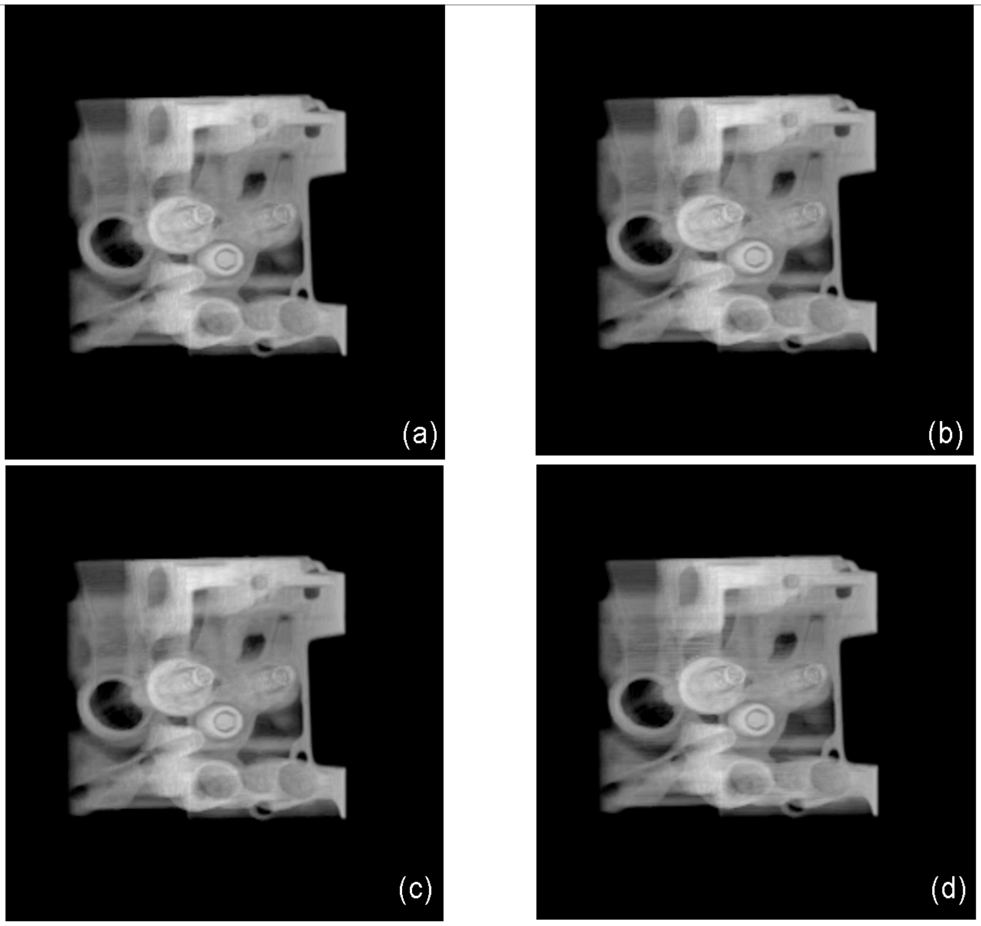


Figure 6.11: The influence of encoding error.Image a is the reference image rendered without coherence encoding.Image b,c,d are images rendered with coherence encoding using an error threshold value respectively equal to $0.2, 0.4$ and 0.5.

6.3 Accelerating Ray Casting

6.3.1 Existing ray casting acceleration methods

Ray casting is known for being the best method for rendering high quality images. Unfortunately, this process is time consuming and very expensive. This is due to the huge number of rays to be shot and the exhaustive computation needed for trilinear interpolation when resampling and Phong shading calculation. Thus, many strategies have been proposed, over the two last decades, to accelerate this algorithm. For this purpose, exploiting coherence is undoubtedly, most often, the key to achieve fast and efficient ray-casting. Based on that, different strategies have been proposed that can be classified into two groups. On one hand side, a group which aims to minimize the number of rays to be casted; this can be realized either by adaptive sampling [35, 36, 37], i.e., reconstructing an image with less samples in the image plane, or by decreasing the amount of rays to be casted for each sample e.g shadows caches [43]. On the other hand, a group whose goal is to improve the core of the ray-casting algorithm such that the rays can be processed faster, to accomplish this objective, a different number of data structures has been used to speed up this process[18] such as Bounding Volume Hierarchies (BVH), grids, octrees and Binary Space Portioning (BSP)[23]. Recent work has focused on kd-trees [13]. In [59], Wald et al exploit coherence by processing packets of rays in parallel. Using this approach, they decrease the algorithm computational time using SIMD instructions on multiple rays in parallel, reduce memory bandwidth by requesting data only once per packet and increase cache utilization at the same time. Exploiting the coherence in a packet of rays has yield further improvement in Multi-Level ray tracing algorithm [46], where a bounding frustum drives the kd-tree traversal of rays in bulk instead of considering each ray separately. While those techniques, namely ray packets, frustum testing, were only available for unstructured grids, they were recently extended to uniform grids by [58]. Our goal in this chapter is to take advantage from spatial coherence on uniform grid by exploiting the coherence regions in a semi-transparent objects.

6.3.2 Accelerating Ray-Casting by Coherence Encoding

We have seen in the previous section that we can accelerate shear warp algorithm by exploiting the voxel's opacity linearity in voxel scanline direction which corresponds to one of the three principal axis. We can extend this principle for exploiting coherence in ray-casting by considering that the voxel values change linearly in all directions. This allows to reduce the sampling rate along the ray by using a coherence distance which corresponds to the interval of coherence along the ray [5]. As shown in the figure 6.12 by [5], inside the linear segment, i.e., the coherent region of the ray path, the normals are constant; therefore, instead of sampling the ray between x_0 and $x_0 + d$, we only have to consider the voxel values at the boundaries. In fact, for opacity estimation during rendering, instead of calculating the opacity value for each sample in the linear segment using the trilinear interpolation, we save time by only achieving linear interpolation between the two boundary voxels. This, on one hand, on the other hand, for evaluating the Phong shading model, the fact that the normals are constant inside those coherent regions is helpful. In fact, the Phong shading depends on three vectors namely the light vector $\overrightarrow{L}$, the normal vector $\vec{N}$ and the viewer direction $\vec{V}$. When the light source can be considered far enough and since the view direction is the same for a given ray as well as the normal inside the linear

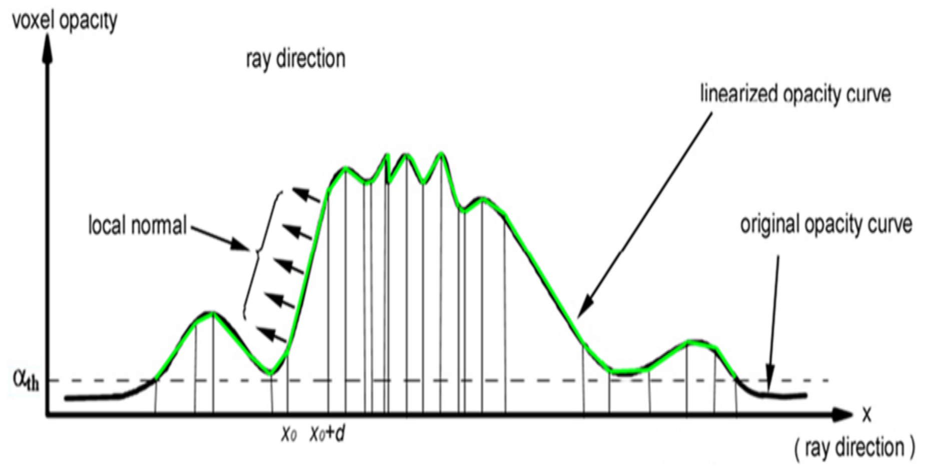


Figure 6.12: Approximating the voxel opacity value curve by piecewise linear segments

segment, then the shading value for this coherent region is the same, by consequence it can be assimilated to one of the boundary shading values. This allows to save many operations for shading calculation which is one of the most computational tasks in ray-casting rendering process. The coherence is encoded in shear warp algorithm by only considering the marching direction of the voxel traversal, thus, the view dependence for coherence encoding is no problem because when rendering, the volume is always shared and composited parallel to one of the three major axes. However, encoding coherence for ray-casting is problematic due to the arbitrary directions of the viewing rays casted through the volume. Since the encoding coherence must be view independent, a minimum coherence distance among all the possible view directions, which pass through each voxel, have to be calculated and stored (see figure 6.13). During rendering, when a ray passes through a voxel, the related distance

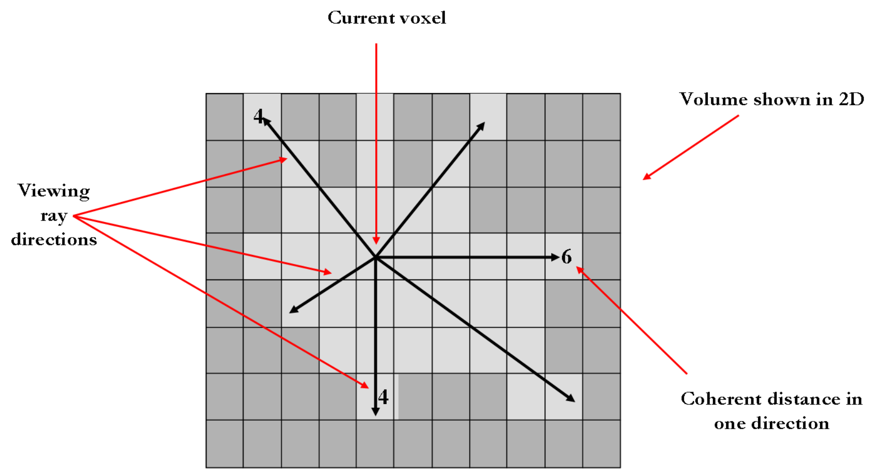


Figure 6.13: Coherence encoding for ray-casting: for each voxel, the shortest coherent distance is fetched among all possible directions and saved

coding is read out and used to determine the position of the next sample point which corresponds to the limit of the coherent region along the ray. Thus, the shading calculations for the samples in between are minimized. Chen [5] proposed a linear interpolation to get the information for the samples inside the coherent region, while we use the pre-integrated lookup tables to get directly the opacity and color inside the coherent region of a certain length. For efficiency's sake, the local coherence distance value is stored in a 3D array which is of the same size as the volume, so that both the coherence distance and the voxel access have the same addressing. In our algorithm, since we are using the `VGLSampleGridData` type to save the volume data set, a new grid of the same type and size is created to store the distance coding for each voxel in the volume. The coherent encoding is in fact similar to the distance coding for space leaping. The later is also encoded view independently with only difference that we consider empty regions instead of coherent regions which is a special case of coherence. The distance from each empty voxel to the nearest non-empty voxel is fetched by approaches like the two pass morphological filtering algorithm [27] and saved in a 3D distance array. These distances values are later used in the ray-casting algorithm to skip empty spaces. For calculating the coherence distance encoding (EDC), there exist different strategies. The brute force EDC searches all possible ray directions for the shortest coherence distance. This method was showed as being too expensive due to the huge number of considered ray directions. An improvement of this algorithm was given by [6], considering only 26 searching directions instead of all possible ray directions. These directions correspond to the connecting directions between a voxel and its 26 neighboring voxels, which are either axis parallel or diagonal. This algorithm realized much better performance than the brute force EDC algorithm but the image quality was not guaranteed because of the risk of missing tiny details when the encoded coherence distance increases. Chen [5] addressed this problem by using the Taylor expansion based EDC, described in the following.

6.3.2.1 The Taylor Expansion based Extended Distance Coding (EDC)

The main idea of ray-casting rendering with coherence encoding is to assign for each voxel in the volume, a view independent maximal spatial coherence distance. That means, along any ray direction, the integral of the difference between the opacity curve and its approximation by a line is less then a predefined error threshold ε. This leads to the following formula:

$$\begin{aligned} d &= max\{l \in \mathbb{R}^+ | \int_0^l \mid f(x\vec{L}) - (ax+b) \mid dx \leq \varepsilon, \\ \forall \vec{L} &\in \mathbb{R}^3, \|\vec{L}\| = 1, a = \frac{f(l\vec{L}) - f(0)}{l}, and\ b = f(0)\} \end{aligned} \tag{6.3.1}$$

This formula is depending on both the considered direction and the size of the linear domain. Considering each ray direction, which passes through each voxel in the data, is problematic due to the exhaustive combination checking of view directions. Thus, it would be interesting if the calculation of the encoded distance could be expressed independently of the view direction. This can be achieved by applying the Taylor expansion to formula 6.3.1. For simplicity, we first consider f as 1D function. In 6.3.1, a can be approximated by the local gradient which is given by the first order differential value of the function f. According to the Taylor formula [20], if $f \in \mathcal{C}^2(\mathbb{R})$ is defined on a bounded interval, then f can be approximated, in the local neighborhood of $x = 0$, by a linear function:

$$f(x) = f(0) + xf'(0) + R(x) \tag{6.3.2}$$

With the remainder

$$R(x) = \frac{x^2}{2} f"(\xi), \xi \in [0, x]. \tag{6.3.3}$$

Approximating the parameter a in 6.3.1 by $f'(0)$, we obtain:

$$|f(x\vec{L}) - (ax + b)| = |f(x\vec{L}) - (x \cdot f'(0) + f(0))| = \frac{x^2}{2} |f"(\xi)| \tag{6.3.4}$$

In addition

$$\exists x_{max}, \quad \forall \xi \in [0, x], |f"(\xi)| \leq |f"(x_{max})| \tag{6.3.5}$$

Thus, 6.3.1 can be rewritten with:

$$d = max\{l \in \mathbb{R}^+ | \int_0^l \frac{x^2}{2} |f"(x_{max})| dx \leq \varepsilon\} \tag{6.3.6}$$

Here, in 6.3.6, the coherence distance does not depend anymore on the ray directions. It is only determined by the maximal second-order differential value and the size of the coherent region. Thus, given 6.3.6, the encoded coherence can be calculated as follows:

$$Error(l) = \int_0^l |R(x)|dx \leq \int_0^l \frac{x^2}{2}|f"(x_{max})|dx = \frac{l^3}{6}|f"(x_{max})|. \tag{6.3.7}$$

For a predefined user coherence error Err_{lin}, this leads to:

$$Error(l) \leq \frac{l^3}{6}|f"(x_{max})| \leq Err_{lin}. \tag{6.3.8}$$

Therefore the maximal coherence distance is given by:

$$l \leq \sqrt[3]{\frac{6Err_{lin}}{|f"(x_{max})|}} \tag{6.3.9}$$

We then extend the result for 3D functions, we obtain:

$$l \leq \sqrt[3]{\frac{6Err_{lin}}{|f_{xx}"(\vec{x_{max}}) + f_{yy}"(\vec{x_{max}}) + f_{zz}"(\vec{x_{max}}) + f_{xy}"(\vec{x_{max}}) + f_{xz}"(\vec{x_{max}}) + f_{yz}"(\vec{x_{max}})|}}$$

Thereby, the local coherence error depends only on the predefined user linearization error Err_{lin} and the second order difference value at each voxel.

6.3.3 Implementation

Before rendering volume with ray-casting based coherence encoding, we first compute and save the coherence encoding distances for each non-empty voxel, using the Taylor Expansion method described above. The algorithm for computing the coherence distance is mainly composed of three steps. First, we evaluate for each non-empty voxel in the volume, its maximum local coherent distance, which is its distance to the nearest empty voxel on its neighborhood. We then, save it in a distance grid with the same size as the volume. Second, for each non-empty voxel, we compute according to equation 6.3.10, the candidate coherence distance obtained by the maximum local coherence distance:

$$l(x,y,z) \quad := \quad \begin{cases} \sqrt[3]{\frac{6Err_{lin}}{|f"(x,y,z)|}} & \text{if} \quad f"(x,y,z) \neq 0 \\ maxdis & \quad \text{otherwise} \end{cases} \tag{6.3.10}$$

where

$$f"(x,y,z) = f_{xx}"(x,y,z) + f_{yy}"(x,y,z) + f_{zz}"(x,y,z) + f_{xy}"(x,y,z) + f_{xz}"(x,y,z) + f_{yz}"(x,y,z)$$

The resulted values are saved in a temporary buffer with the same size as the volume. Finally, a flood-fill process is used to compare, for each non-empty voxel, its candidate coherence distance value l read from the buffer, with all distance values d on the 3D-l neighborhood. Then whenever d is higher than l, d is putted to l, otherwise the original value is preserved. The pseudo-code for this algorithm is showed in algorithm 4. To convenably store the coherence distances for each voxel in the volume data, it is necessary to use the same data structure used to save the volume, so that we can access each voxel and its corresponding coherence distance with the same coordinates. For this purpose, we adopt the `VGLSampleGridData` class provided by the VGL library. We first create a class *SampleGridCoherence* (see figure 6.14) which allows us to calculate the coherence distances for each voxel in the volume and save them into a 3D data structure for further access. Then, we apply the coherence encoding technique to the ray-casting rendering algorithm. In the previous work of Chen [5], encoding coherence during rendering consisted on reading color and opacity information for both voxels at the boundary of the coherence sample interval, then for each sample in between calculating shading information by linearly interpolating between the boundary voxels and apply compositing. In our algorithm, we will improve this part by applying the pre-integration technique to the rendering

Algorithm 4 Tayloredc: The pseudo-code for the Taylor expansion based EDC algorithm: the input to this algorithm is a classified discrete volume data set *vol*. The output is a volume data set having the same size as *vol* and containing the coherence encoding distances.

```
{allowed mistake}
$static\ float\ ErrMaxLin = ERRMAX$;
{first step: get the distances of all non-transparent voxels to the nearest transparent one, using the same algorithm like in the space leaping part of the calculation}
{fill the matrix with maximal values but only non-transparent voxels}
$maxvalue(vol, maxdist, material)$;
{get the distance values for all non-transparent voxels}
$caldis(vol, material, mindist)$;
{check the distances of all non-transparent voxels to the borders of the volume file}
$borders(vol, material)$;
{second step: Formula 6.3.10}
{a temporaryDistanceBuffer is used in this step.}
VGLSampleGridData $tmpDisBuffer($VGL_TYPE_UINT8, VGLSampleGridSize$(dx, dy, dz))$;
{third and last step: merge the two distance matrixes}
$merge(tmpDisBuffer, vol, material)$;
return $true$;
```

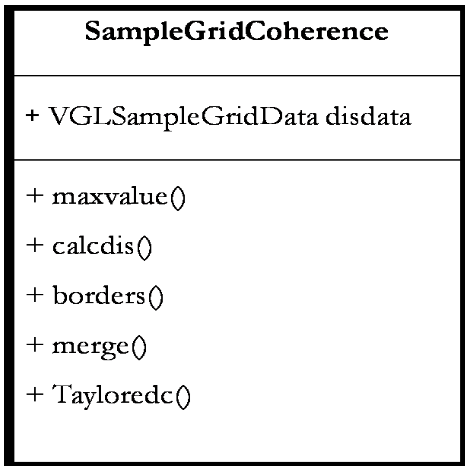


Figure 6.14: The class SampleGridCoherence for encoding coherence distances

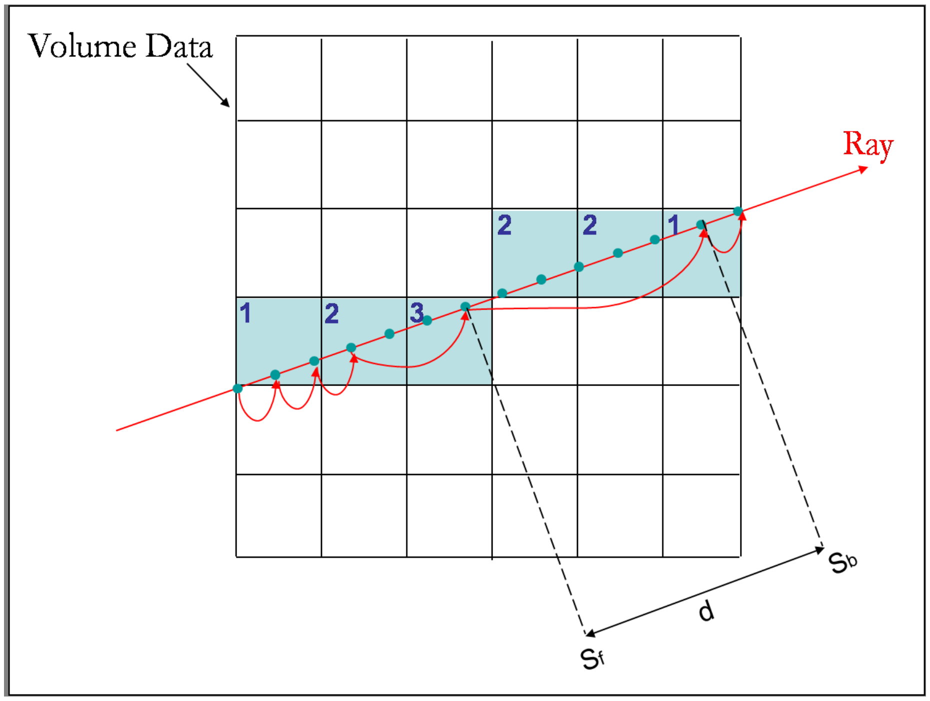


Figure 6.15: Ray-casting algorithm with Coherence Encoding and Pre-Integration

process as shown in figure 6.16. Instead of linearly interpolating sampling in the coherence interval and compositing them one by one, thanks to the pre-computed pre-integration lookup tables, we only have to read the gray values at the front and back samples of the coherence interval, as well as the coherence distance, and plug them into the LUT to get directly the shading results for this interval (see figure 6.15). This is possible, since as we explained in the last chapter, we precompute the LUT for different distances.

In the following, algorithm 5 shows the pseudo-code for the method $EncodeCoherence()$ mentioned in the figure 6.16.

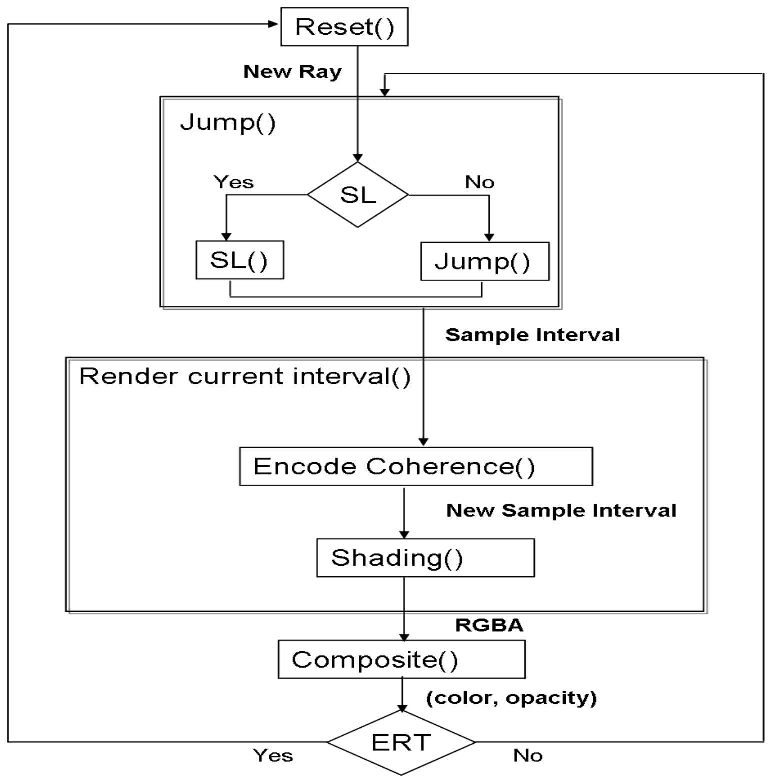


Figure 6.16: Rendering Process

Algorithm 5 Encode Coherence: Encoding Coherence Method

```
$end = ray.end;$
$t = ray.current[0];$
$distance = SampleGridCoherence.disdata[t];$
**if** $distance > 2$ **then**
   $t+ = distance * ray.rayOneVoxelDistance$
   **if** $t \geq end$ **then**
      $t = end;$
   **end if**
   $ray.current[1] = t;$
**end if**
```

6.3.4 Results and Analysis

We experimented the ray-casting algorithm with two acceleration methods: first using only coherence encoding and second combining both coherence encoding and pre-integrated volume rendering technique. We compared the performances and image qualities between both approaches. All experiments were carried out by using half of the voxel size as the sample unit to avoid artifacts as suggested by [32]. In order to find what kind of data type is best accelerated by our methods, we considered two classification types namely opaque and semi-transparent objects. As data set for our experiment, we chose tow engine data set of size respectively $256 \times 256 \times 128$ voxels and $128 \times 192 \times 256$ voxels, both are 8 bit data. To compare the performance we use two metrics: the rendering time and the percentage of samples saved. While the rendering time is computer platform dependent, the samples saved are not. The percentage of samples saved is given by the following formula:

$$SamplesSaved(\%) = \frac{Totalsamples - Newsamples}{Totalsamples} * 100 \qquad (6.3.11)$$

where $Totalsamples$ is the initial samples number and $Newsamples$ is the number of samples after applying encoding coherence method.

Table 6.1 contains the experimental results for the two acceleration methods. The results are obtained by using the opacity transfer function that makes the volumes opaque. The rendering times listed in table 6.1 show that for both algorithms: coherence encoding that we later reference to as technique I and pre-integrated rendering combined with coherence encoding that we later reference to as technique II, the rendering time is a decreasing function of the error tolerance and the number of saved

Image No	Dataset + technique	Error Tolerance	Saved Samples (%)	Rendering Time (s)
6.17.a		0	$0,001$	$1,392$
6.17.b	Opaque Engine CT	5	$0,381$	$1,382$
6.17.c	$128 \times 192 \times 256\_8bit$	5.5	$1,050$	$1,372$
6.17.d	+ coherence encoding	6	$1,058$	$1,352$
6.18.a	Opaque Engine CT	0	$3,124$	$3,9$
6.18.b	$128 \times 192 \times 256\_8bit$	1	$3,921$	$3,8$
6.18.c	+ coherence encoding	1.5	$6,352$	$3,4$
6.18.d	+ pre-integration	2	$9,470$	$3,2$

Table 6.1: Rendering parameters for opaque data sets rendered with our algorithm

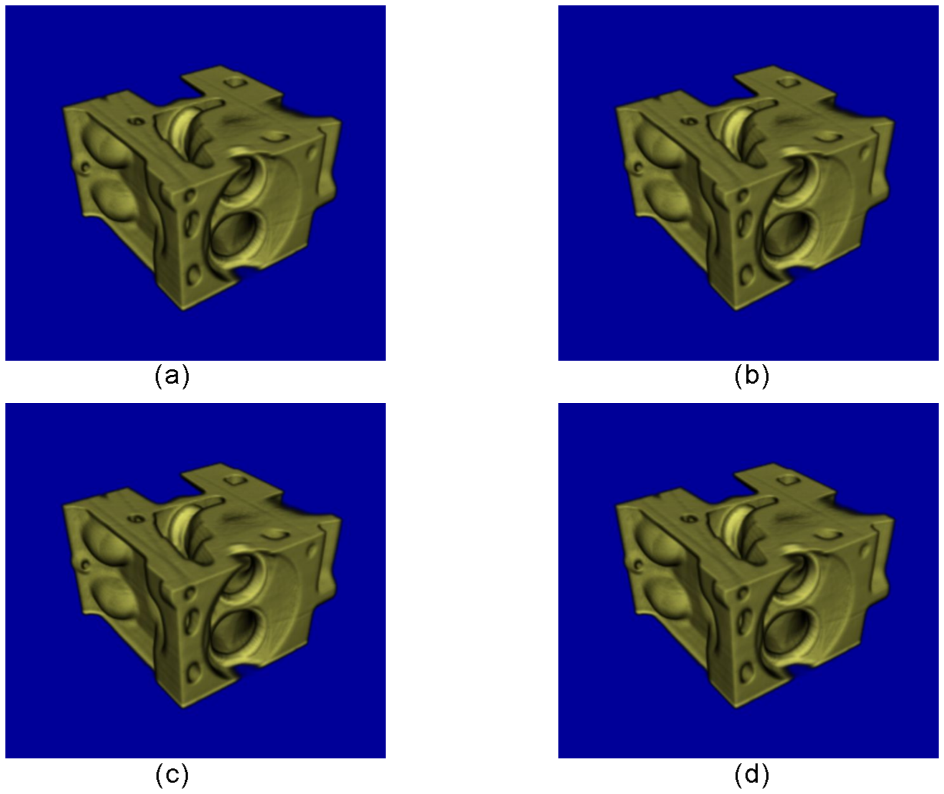


Figure 6.17: Rendering opaque engine with coherence encoding

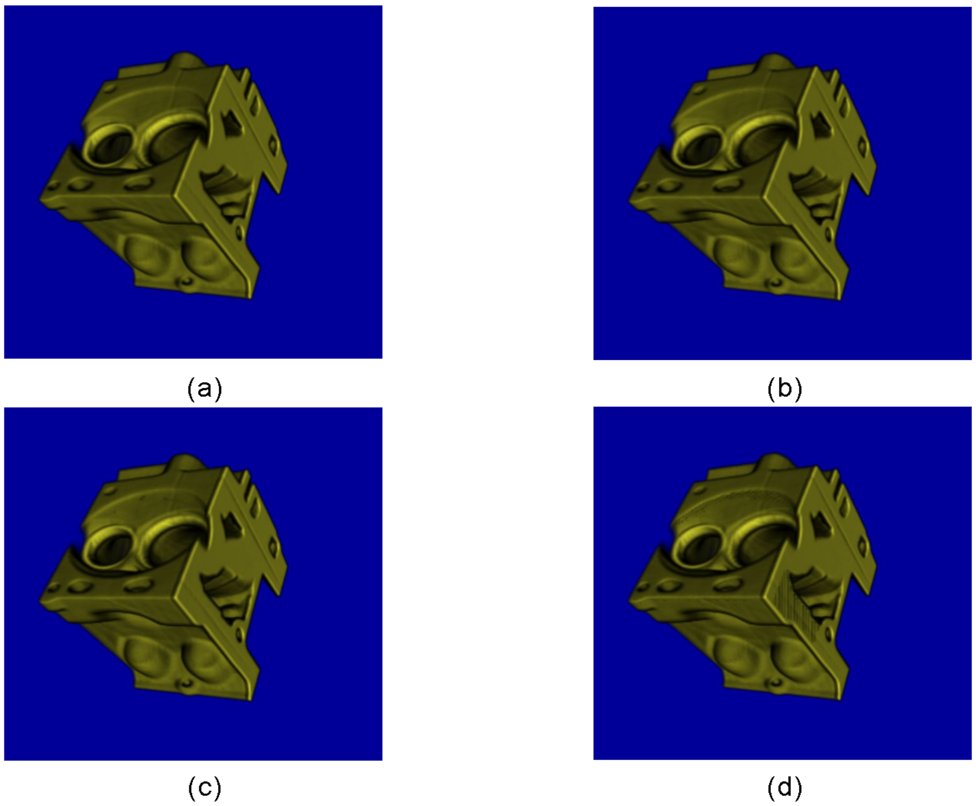


Figure 6.18: Rendering opaque engine with coherence encoding and pre-integrated technique

Image No	Dataset + technique	Error Tolerance	Saved Samples (%)	Rendering Time (s)
6.19.a	Semi-Transparent Engine CT $256 \times 256 \times 128\_8bit$ + coherence encoding	0	$3,127$	$16,6$
6.19.b		5	$3,228$	$15,8$
6.19.c		6	$3,242$	$14,73$
6.19.d		7	$3,256$	$14,25$
6.20.a	Semi-Transparent Engine CT $256 \times 256 \times 128\_8bit$ + coherence encoding + pre-integration	0	$2,823$	$5,789$
6.20.b		5	$2,834$	$5,538$
6.20.c		10	$2,875$	$5,288$
6.20.d		14	$2,908$	$5,227$

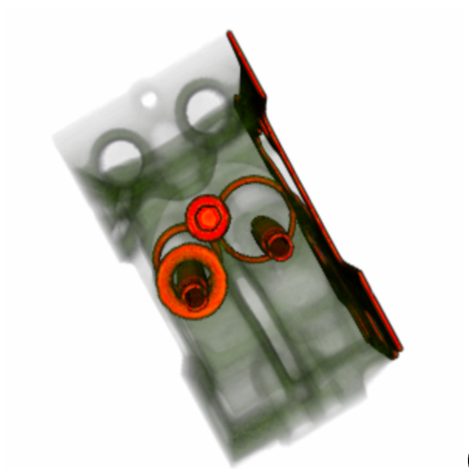
Table 6.2: Rendering parameters for semi-transparent data sets rendered with our algorithm

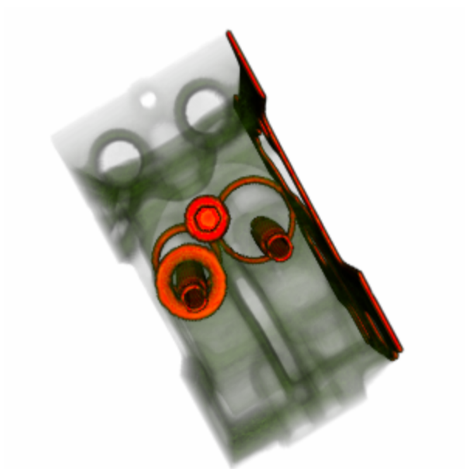
samples is an increasing one, as illustrated in figure 6.21.a and 6.21.c. When comparing between the results of the two rendering techniques for opaque data set, unlike one can expects, the rendering time in technique II is slower than in technique I even the use of the pre-integrated volume rendering technique. This is in fact due to two reasons: first, the bilinear interpolation to obtain the color and opacity results and second, the presence in the shading function of a function *lookdis* (see procedure 6) which have to look for the corresponding input to the pre-integrated lookup tables as well as the values to interpolate in between. Thus, since for the opaque data set, the presence of coherent regions is rare, the consuming time due to those two factors is not attenuated.

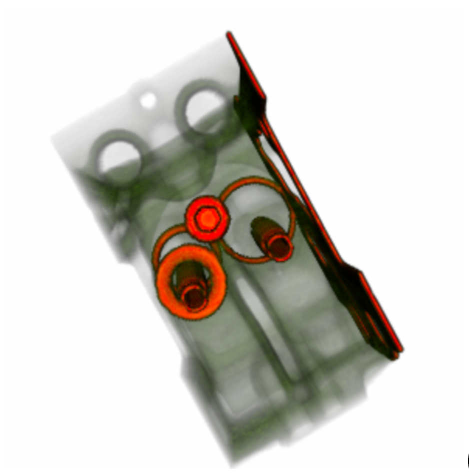
Procedure 6 lookdis($float\ x, float\ y, float\ d, int\ \delta i$): The lookdis procedure

```
$i = log(d) * ln2;$
$x = P2[i];$
$y = P2[i+1];$
```

Table 6.2 contains the experimental results for the technique I an II. The results


(a)


(b)


(c)

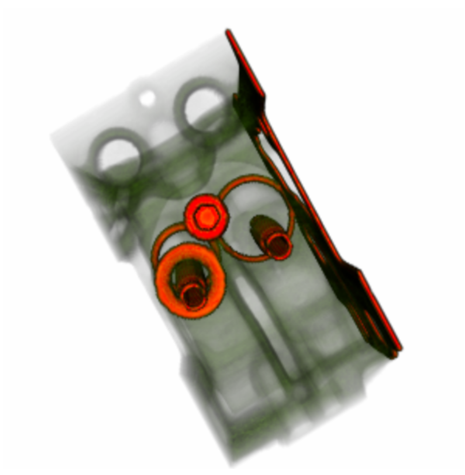
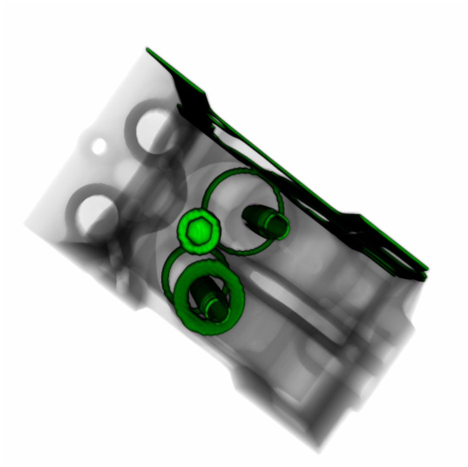
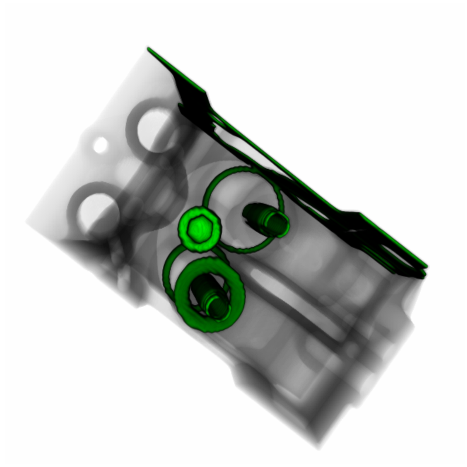
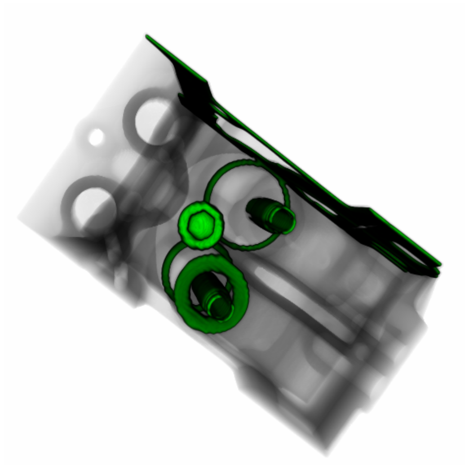

(d)

Figure 6.19: Rendering semi-transparent engine with coherence encoding


(a)


(b)


(c)

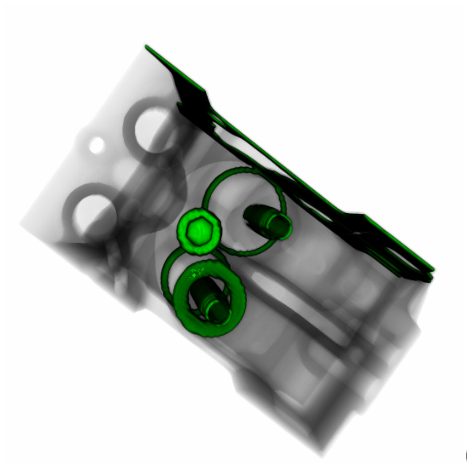
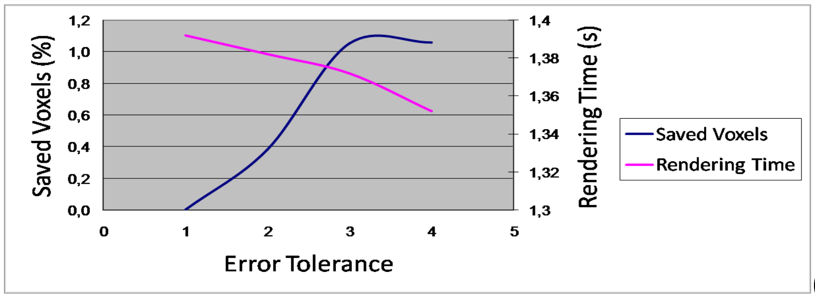
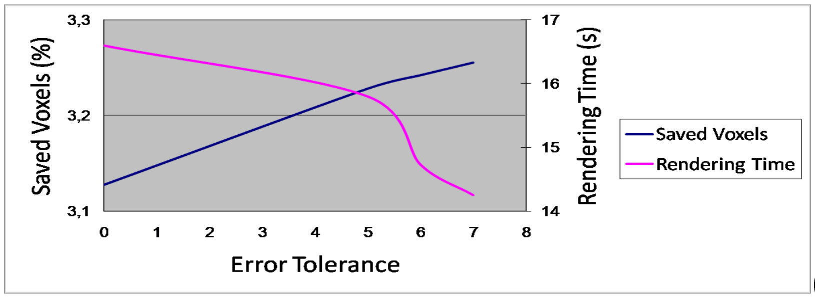

(d)

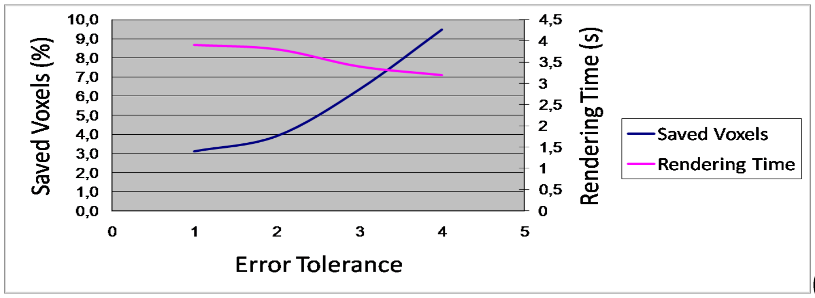
Figure 6.20: Rendering semi-transparent engine with coherence encoding and pre-integrated technique



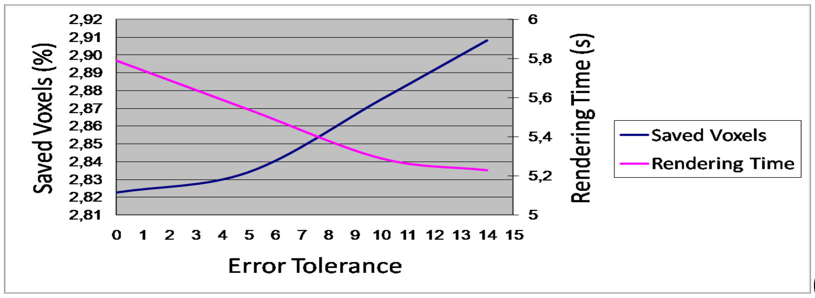
(a)



(b)



(c)



(d)

Figure 6.21: Influence of encoding error on rendering performance for different techniques and data sets: (a)opaque engine(128 × 192 × 256 voxels) rendered using only coherence encoding, (b)semi-transparent engine(256 × 256 × 128 voxels) rendered using only coherence encoding, (c)opaque engine(128 × 192 × 256 voxels) rendered using coherence encoding and pre-integration technique, (d)semi-transparent engine(256 × 256 × 128 voxels) rendered using coherence encoding and pre-integration technique

are obtained by using the opacity transfer function that makes the volumes semi-transparent. The rendering times listed in table 6.2 show that for both algorithms, the rendering time is a decreasing function of the error tolerance and the number of saved samples is an increasing one, as illustrated in figure 6.21.b and 6.21.d. Unlike the case where the data set was opaque, we notice here that our algorithm is accelerated by the pre-integrated technique by factor of approximatively 3. This is due to the larger coherent distance present during shading.

6.4 Conclusion

In this chapter, we first implemented the shear warp algorithm with coherence encoding. We further applied the coherence encoding acceleration technique to the ray-casting algorithm: in a first step, we implemented using the VGL, the Taylor expansion based extended distance coding, which calculates for each voxel in the volume a view independent distance coherence. In a second step, we reimplement the ray-casting algorithm process accelerated by coherence encoding as well as pre-integrated volume rendering technique and we achieved a speed up of factor 3 for semi-transparent objects due to the large coherent regions they present.

Chapter 7

Rendering Deformed Objects

7.1 Introduction

In this chapter, we present a method to render deformed objects. We first introduce the previous work related to this field, then we give an overview about the mathematical background of the deformation tools used further in our implementation, precisely the Free Form Deformation (FFD). Afterwards, we describe how ray casting is applied to deform object in 3D space. Later, we present the changes and the optimizations provided to the current implementation state. Finally we show the corresponding results.

7.2 Related Work

Different approaches for describing volume deformation involves volume morphing [12, 25, 28, 54], a technique used to generate smooth object transformation. This deformation technique is however not automatic. It consists on selecting landmarks in both original and target object, which are used later to calculate the transformation. Another deformation techniques are the geometrically based deformation model and the physically based deformation model. In the following, we shortly mention first some models for physically based deformation, then we describe in more details the previous work in geometric based deformation. In physically based deformation, a physical model is applied for the full object. The computations can include spring-like models [45, 15], continuum models [16], finite element methods [7] or landmark deformations [11]. In geometrically based deformation, typical volume deformation consists on deforming the object in a first step and then rendering the deformed object. Barr [1] did the first work on object deformation. He developed new hierarchical solid modeling operations, which simulate 3D transformations of geometric objects. He found out that the normal vector of an arbitrarily deformed smooth surface can be calculated directly from the surface normal vector of the undeformed surface and a transformation matrix, and call this the normal vector transformation rule. The limitation to his technique is that it is not very suitable for realize complex deformation schemes. For deformation in ray casting algorithm, he proposed the idea of inverse ray deformation which consists on combining both deformation procedure and rendering process. As shown in figure 7.1, the deformation can be achieved either by intersecting the deformed primitive by the incident ray, or by intersecting the undeformed primitive by the twisted ray. This idea was further adopted by Kurzion and Yagel [33]. They introduced the concept of ray deflectors for deforming volume data sets during the rendering stage by deforming viewing rays using deflectors. A deflector is a gravity vector, positioned in space, that bends all rays passing through its defined area of influence as shown in figure 7.2. In order to generate a local deformation, all sight rays are transformed in a direction opposite to that of the desired visual effect.

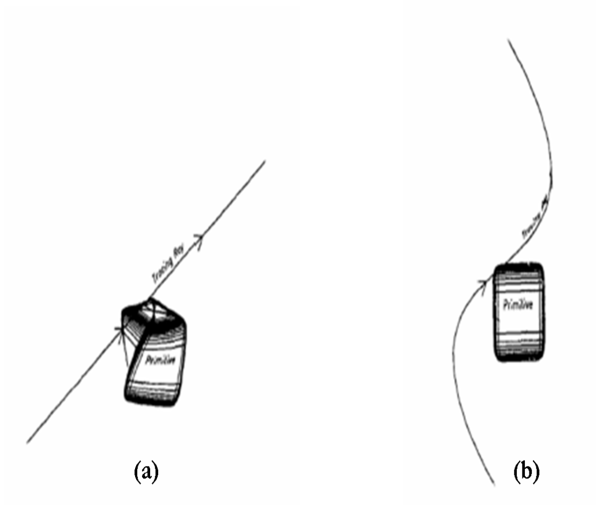


Figure 7.1: Deformation by inversely deforming rays. (a) Deformed primitive, in undeformed space. (b) Undeformed primitive, in its undeformed coordinate system, showing path of ray (By Barr [1])

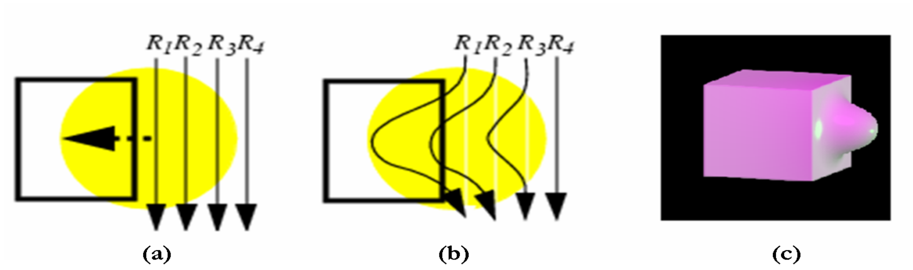


Figure 7.2: Ray deflector. (a) Locate a ray deflector at the right side of a hexahedron (2D draft). (b) The ray trajectories are deformed within the ray deflector. (c) The visual effect. (By Kurzion [33])

Later, Chen [24] used the inversely deformed rays to produce the expected deformation effect by using a uniform spline grid based free form deformation scheme. In our object deformation algorithm, we will also use inversely deformed rays by integrating some optimizations to Chens work.

7.3 Mathematical Background

7.3.1 Parametric representation of objects

The easiest way to deform an object is to use a parametric representation. In order to change the shape of a parametric curve, it's sufficient to move one or several of its control points. The original and the deformed curve have then the same resolution.

7.3.2 B-splines

Given $m+1$ knots t_i with $t_0 \leq t_1 \leq \ldots \leq t_m$, a B-spline curve of degree p (order $k = p+1$) can be defined in terms of a set of control points $\vec{P_i}(i = 0, 1, 2...n)$ such as $m = n + p + 1$. In essence, each control point influences the shape of a local part of the curve. The curve is defined by the following equation:

$$\tilde{\mathbf{S}}(t) = \sum_{i=0}^{n} \tilde{\mathbf{P}}_i b_{i,p}(t) \ , \, t \in [t_0, t_m] \tag{7.3.1}$$

Basically, this is a summation of the blending function $b_{i,p}(t)$ and the control points $\vec{P_i}$. The blending function of degree p can be defined using the Cox-de Boor recursion formula:

$$\begin{aligned} b_{i,0}(t) &:= \begin{cases} 1 & \text{if} \quad t_i \le t < t_{i+1} \\ 0 & \quad \text{otherwise} \end{cases} \\ b_{i,p}(t) &:= \frac{t - t_i}{t_{i+p} - t_i} b_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} b_{i+1,p-1}(t). \end{aligned} \tag{7.3.2}$$

Due to the spline locality property, the equation (7.3.1) gives:

$$\tilde{\mathbf{S}}(t) = \sum_{i=i_0-p}^{i_0} \tilde{\mathbf{P}}_i b_{i,p}(t) \ , \ t \in [t_{i_0}, t_{i_0+1}] \tag{7.3.3}$$

7.3.2.1 The knot vector

We distinguish three types of knot vectors. First, the *Uniform* knot vectors: these are knot vectors for which

$$t_{i+1} - t_i = \text{constant}, \forall i$$

For example:

$$\begin{aligned} &[1, 2, 3, 4, 5, 6, 7, 8] \\ &[0, 1, 2, 3, 4, 5] \\ &[0, 0.25, 0.5, 0.75, 1.0] \end{aligned}$$

Second, the *Open Uniform* knot vectors: these are uniform knot vectors which have k equal knot values at each end:

$$\begin{aligned} t_i = t_1, &\quad i \le k \\ t_{i+1} - t_i = \mathit{constant}, &\quad k \le i < n + 2 \\ t_i = t_{k+(n+1)}, &\quad i \ge n + 2 \end{aligned}$$

For example:

$$\begin{aligned} &[0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4] && (k = 4) \\ &[1, 1.......1, 0.1, 0.1, 0.1, 0.3, 0.5, 0.7, 0.7, 0.7, 0.7, 0.7] && (k = 5) \end{aligned}$$

And finally the *Non-uniform* knot vectors. This is the general case, the only constraint being the standard $t_i \le t_{i+1}, \forall i$. For example:

$$[1, 3, 7, 22, 23, 23, 49, 50, 50]$$
$$[1, 1, 1, 2, 2......, 6, 6, 7, 7, 7]$$
$$[0.2, 0.7, 0.7, 0.7, 1.2, 1.2, 2.9, 3.6]$$

7.3.3 De Boor Algorithm

The de Boor's algorithm, described in this paragraph, is a procedure which efficiently evaluates the expression for equation (7.3.3). It provides a fast and numerically stable way for finding a point on a B-spline curve. The algorithm is described below:

$$Given \begin{cases} u \in [u_{i_0}, u_{i_0+1}] \\ and \\ P_{i,0} = \vec{P}_i, \forall i = i_0 - p, ..., i_0 \end{cases}$$

We calculate

$$P_{i,k} = (1 - \alpha_{k,i})P_{i-1,k-1} + \alpha_{k,i}P_{i,k-1}; \qquad k = 1, \ldots, p; \quad i = i_0 - p + k, \ldots, i_0$$

with

$$\alpha_{k,i} = \frac{u - u_i}{u_{i+p+1-k} - u_i}.$$

Then

$$S(u) = P_{i_0,p} \tag{7.3.4}$$

The de Boor algorithm is illustrated by the figure 7.3.

7.3.4 Tri-variate Tensor product B-spline

Tri-variate Tensor product B-spline is simply an extension of B-spline curve to the $3D$ space, and for a degree p in each parametric axis, and over a knot sequence

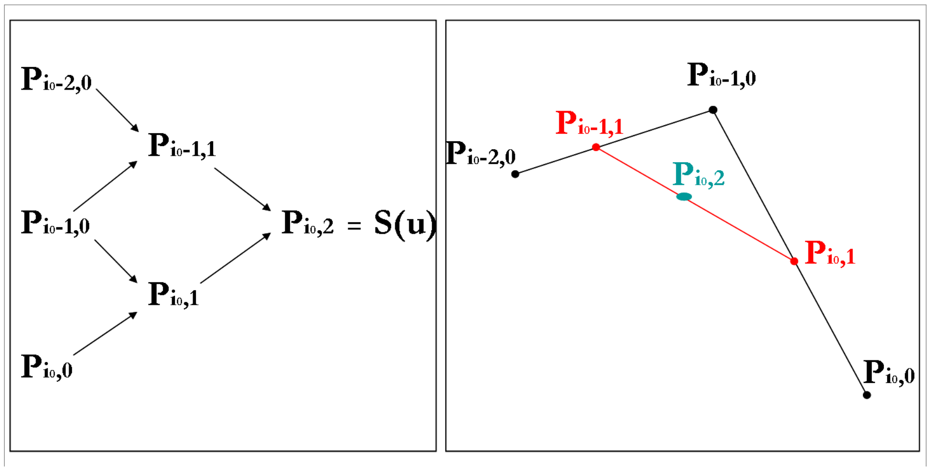


Figure 7.3: The De Boor Algorithm for a quadratic B-Spline segment

$U = \{u_i\}, V = \{v_j\}$ and $W = \{w_k\}$, is defined by the following equation:

$$F(u,v,w) = \sum_i \sum_j \sum_k P_{i,j,k} B_i(u) B_j(v) B_k(w) \tag{7.3.5}$$

To evaluate this function, we use the de Boor Algorithm. In fact, equation (7.3.5) can be rewritten by:

$$F(u,v,w) = \sum_i B_i(u)[\sum_j B_j(v)[\sum_k B_k(w) P_{i,j,k}]] \tag{7.3.6}$$

Let

$$\begin{aligned} Q_i &= \sum_j B_j(v)[\sum_k B_k(w) P_{i,j,k}], \\ Q_j &= \sum_k B_k(w) P_{i,j,k} \\ \textit{and } Q_k &= P_{i,j,k} \end{aligned} \tag{7.3.7}$$

Then

$$\begin{aligned} Q_i &= \sum_j B_j(v) Q_j \\ \textit{and } F(u,v,w) &= \sum_i B_i(u) Q_i \end{aligned}$$

By consequence, for each i, j, k, we start by storing $P_{i,j,k}$ in Q_k, then evaluating Q_j with the de Boor algorithm as described in 7.3.4, we do the same with Q_i and finally we calculate the result $F(u,v,w)$. This process is illustrated by the pseudo-code given in the algorithm 7.

Algorithm 7 FFD(P, u, v, w): Evaluation of the tri-Variate B-Spline Tensor product by de Boor algorithm

```
{P: the control points array}
{u, v, w are the local coordinates of the current point}
for $i = i_0 - p$ to $i_0$ do
  for $j = j_0 - p$ to $j_0$ do
    for $k = k_0 - p$ to $k_0$ do
      $Q_k = P_{i,j,k}$;
    end for
    $Q_j = Boor(Q_k, B_k(w))$;
  end for
  $Q_i = Boor(Q_j, B_j(v))$;
end for
$F(u,v,w) = Boor(Q_i, B_i(u))$;
return $F(u,v,w)$;
```

7.4 Free Form Deformation (FFD)

The FFD is a method based on B-Splines technique to deform objects. It was first proposed by Slenderer and Parry [53], and it consists on a space mapping in terms of a Tri-Variate tensor product B-splines and the deformation procedure is the following:

1. The object is embedded in a parallelepiped region of space. Then, a local coordinate system is imposed such as for any point $\vec{P}$ inside the parallelepiped:

$$\vec{P} = \vec{O} + u\vec{U} + v\vec{V} + w\vec{W} \tag{7.4.1}$$

   Where $(\vec{O}, \vec{U}, \vec{V}, \vec{W})$ is the local coordinate system. The local coordinate (u, v, w) of $\vec{P}$ are found by the following calculation:

$$\begin{aligned} u &= \frac{(\vec{V} \times \vec{W}) \cdot (\vec{P} - \vec{O})}{(\vec{V} \times \vec{W}) \cdot \vec{U}} \\ v &= \frac{(\vec{U} \times \vec{W}) \cdot (\vec{P} - \vec{O})}{(\vec{U} \times \vec{W}) \cdot \vec{V}} \\ w &= \frac{(\vec{U} \times \vec{V}) \cdot (\vec{P} - \vec{O})}{(\vec{U} \times \vec{V}) \cdot \vec{W}} \end{aligned} \tag{7.4.2}$$

   Thus, $\forall \vec{P}$ in the region, we have: $0 \leq u \leq 1, 0 \leq v \leq 1, 0 \leq w \leq 1$.

2. A grid of control points $\{P_{ijk}, 0 \leq i \leq l, 0 \leq j \leq m, 0 \leq k \leq n\}$ is imposed in the parallelepiped such as:

$$P_{ijk} = \vec{O} + \frac{i}{l}\vec{U} + \frac{j}{m}\vec{V} + \frac{k}{n}\vec{W} \tag{7.4.3}$$

3. For a chosen degree p of B-Splines, the knot numbers can be determined by the formula:

$$m = n + p + 1 \tag{7.4.4}$$

   where m is the knot number, n is the number of control points and p is the curve degree. Thus, the knot vectors can be initialized with:

$$\begin{aligned} \vec{U} &= \{u_0, u_1, u_2, ..., u_{l+p+1}\} \\ \vec{V} &= \{v_0, v_1, v_2, ..., v_{m+p+1}\} \\ \vec{W} &= \{w_0, w_1, w_2, ..., w_{n+p+1}\} \end{aligned} \tag{7.4.5}$$

In our case, we use Uniform Quadratic B-splines which is sufficient to achieve C^1 continuity.

4. To deform the object or part of it, the following steps are necessary:

   - To move the control points to their new position: $P_{ijk} \longrightarrow P'_{ijk}$, according to a transformation function.
   - To determine the local coordinate (u, v, w) of a given point P and its knot index (i_0, j_0, k_0) such that:

   $$u \in [u_{i_0}, u_{i_0+1}], v \in [v_{j_0}, v_{j_0+1}], w \in [w_{k_0}, w_{k_0+1}]. \tag{7.4.6}$$

   - To calculate the coordinate of the deformed point by the following formula:

   $$P_{FFD}(u, v, w) = \sum_{i=i_0-p}^{i_0} \sum_{j=j_0-p}^{j_0} \sum_{k=k_0-p}^{k_0} P'_{ijk} B_i(u) B_j(v) B_k(w). \tag{7.4.7}$$

   Where $B_i(u)$,$B_j(v)$ and $B_k(w)$ are quadratic B-spline basis function. We notice that moving a control poin has only a locally effect based on the B-spline degree, which is an advantage comparing to the original FFD method based on Bzier curve where the deformation was global and then needs more computations.

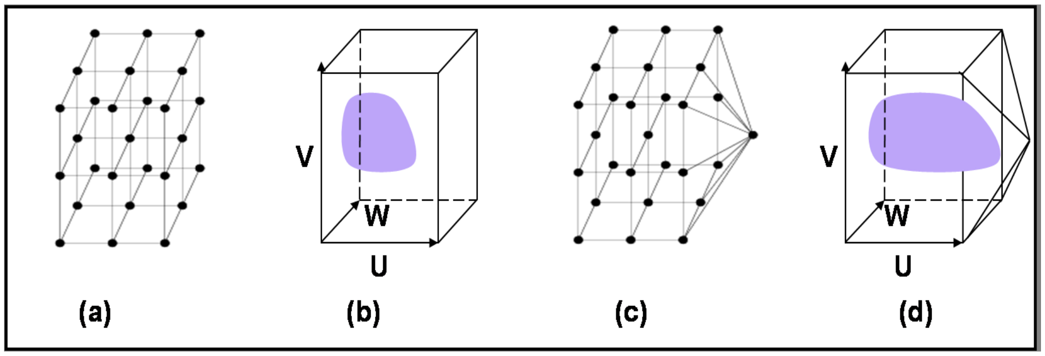


Figure 7.4: FFD deformation. (a)Parallelepiped Grid. (b) Initial position. (c) Deformed FFD grid. (d) Deformed Object

7.5 Ray casting in deformed space

In ray casting, inverse ray deformation is a suited method to produce deformed images by bending the rays casted through the undeformed object. The idea of this technique is straightforward: the volume is embedded into a FFD grid. Then, during rendering, each casted ray is deformed according to the tri-variate tensor product B-splines given in equation (7.4.7), and inversely transformed to its final state. The inverse deformed point set of the ray $P_{inv}(i), i = 0, 1, 2...n$ is defined by:

$$\vec{P}_{inv}(i) = \vec{P}(i) - [\vec{P}_{ffd}(i) - \vec{P}(i)] = 2\vec{P}(i) - \vec{P}_{ffd}(i); \tag{7.5.1}$$

Where $\vec{P}$ is the point on the undeformed ray and $\vec{P}_{ffd}$ its image by the FFD transform on the deformed ray. Therefore, given the inversely deformed ray, this one is approximated by a polyline which is subdivided into polyline segments. Each polyline segment is then handled as a normal ray. In the following, we discuss in details each step of the deformation procedure. In a preprocessing step, we initialize the FFD system: given the size of the 3D object to be rendered, we create a 3D lattice: the

FFD grid which embeds the volume. Then, a local coordinate system is imposed on the FFD grid, as described in equations 7.4.2, a grid of control points is created according to equation 7.4.3, and the knot vectors are initialized as shown in 7.4.6. The number of control points as well as knot vectors are defined by the deformation degree as defined in equation 7.4.4. Further, we define the deformation by moving the control points using a transformation function defined by the user. To implement the above described process, we create a class named *FFD*. Once the FFD system initialized, during the rendering process, whenever a ray is casted, a new domain is created, it consists on the 3D space delimited by the knot points which surround the ray start point. In addition, we know that in this domain, referred to later by $\mathbb{D}$, the deformed ray obtained by the quadratic B-splines basis function, represents a second degree polynomial. Thus, let t_s being the start point of the new ray and $\vec{P_s} = (u_s, v_s, w_s)$ being the corresponding point on the local coordinates system of the FFD grid such as $\vec{P_s} \in \mathbb{D} = [u_{i_0}, u_{i_0+1}] * [v_{j_0}, v_{j_0+1}] * [w_{k_0}, w_{k_0+1}]$. And let t_e being the intersection between the ray and $\mathbb{D}$, i.e. the cubic box limited by the planes: $x = u_{i_0}, x = u_{i_0+1}$ and $y = v_{j_0}, y = v_{j_0+1}$ and $z = w_{k_0}, z = w_{k_0+1}$, and such as $t_e \neq t_s$. Let t_e being represented by $\vec{P_e}$ in the local coordinate system. Given $\vec{P_s}$ and $\vec{P_e}$, we distinguish two cases: if $\vec{P_s}$ is not deformed, then the line segment delimited by $\vec{P_s}$ and $\vec{P_e}$ is not deformed and represents a new polyline segment that is rendered as a normal ray. However, if $\vec{P_s}$ is deformed i.e. one of the 27 control points in its neighborhood is deformed, then the deformed ray embedded into the domain $\mathbb{D}$ is delimited by the points given by the inverse deformation transform of respectively $\vec{P_s}$ and $\vec{P_e}$. Since a spline is a curve that is piecewise p^{th} degree polynomial, therefore, according to equation 7.3.1, this curve must be equal to a polynomial of degree at

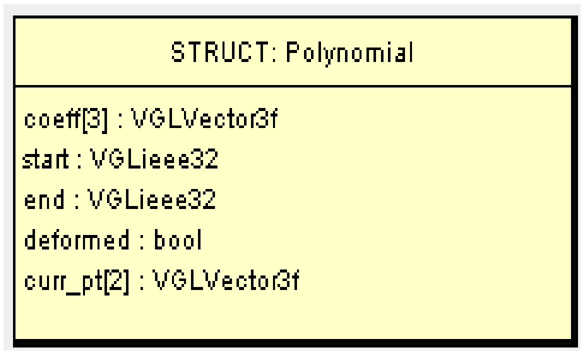


Figure 7.5: The Polynomial data structure to define the domain $\mathbb{D}$

most p. In the next section, we see how to evaluate such polynomial. To encapsulate the information related to each domain $\mathbb{D}$, we create the data structure *Polynomial* whose members are: an array containing the quadratic polynomial coefficients, the start and end point of the domain, situated on the undeformed ray, a flag deformed which indicates if the domain is deformed or not and finally the start and end point, on the 3D space, delimiting the piecewise deformed ray. A description of this structure is shown in figure 7.5 . Thus, the idea is to evaluate explicitly each piecewise quadratic polynomial which defines the deformed ray inside $\mathbb{D}$. For this purpose, the quadratic polynomial's coefficients are determined as described in the following.

7.5.1 Coefficient Polynomial Determination

Let (u, v, w) the corresponding local coordinates of t, a sample point on the ray, then it exists $A, B, C \in R^3$ such as:

$$\begin{aligned} F : [t_s, t_e] &\rightarrow R^3 \\ t &\rightarrow At^2 + Bt + C \end{aligned}$$

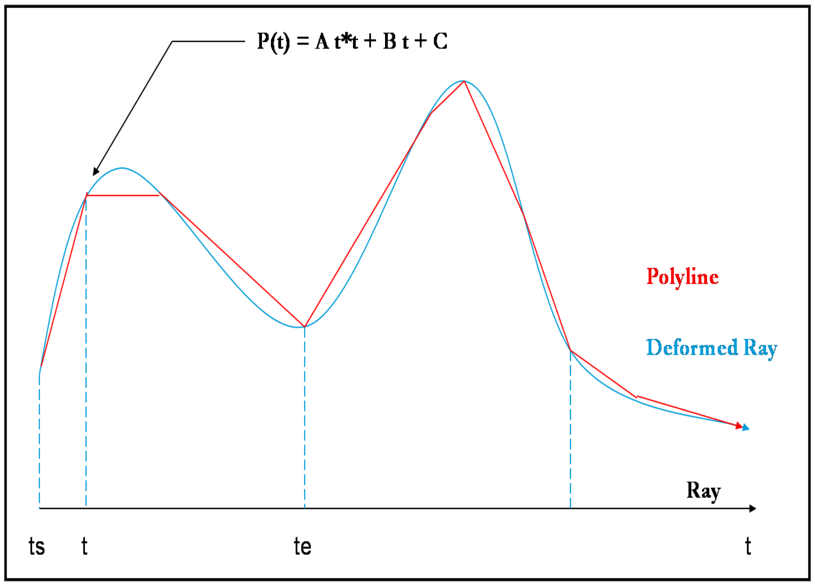


Figure 7.6: Approximation of Deformed Ray by Polyline

We determine A, B, C by solving the following system:

$$\begin{pmatrix} F(t_s) \\ F(t_h) \\ F(t_e) \end{pmatrix} = \begin{pmatrix} t_s^2 & t_s & 1 \\ t_h^2 & t_h & 1 \\ t_e^2 & t_e & 1 \end{pmatrix} * \begin{pmatrix} A \\ B \\ C \end{pmatrix} \tag{7.5.2}$$

Where $t_h = (t_s + t_e)/2$. Then, the obtained quadratic polynomial is approximated by polyline segments as shown in figure 7.6. Therefore, given the quadratic polynomial, our goal is to approximate it by polyline segments to be rendered further. For the polyline segment evaluation, Chen [24] selected a proper point set so that the distance from any point of the continuous ray to the polyline does not exceed a defined threshold distance d, and this, using the curvature based metric. This method consists on assuming $\vec{P}_{n-1}\vec{P}_n$, in figure 7.7, being the polyline segment selected in the last step and then selecting the next polyline segment $\vec{P}_n\vec{P}_{n+1}$ with the longest possible length in order to save the deformation computation for the points in between.

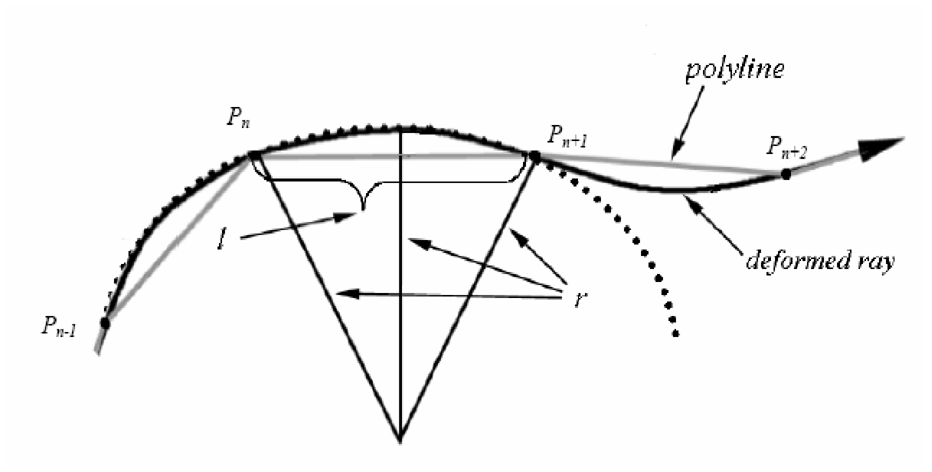


Figure 7.7: Local curvature estimation (By Chen [24])

This technique necessitates the estimation of the local curvature of the deformed ray, by computing two quadratic polynomials interpolating the three points $\vec{P}_{n-1}$,$\vec{P}_n$ and $\vec{P}_{n+1}$. This method requires a loop function which ensures that the length l of a polyline segment can be written as (see figure 7.7):

$$r - \sqrt{r^2 - (\frac{l}{2})^2} \leq d \tag{7.5.3}$$

Thus, l satisfies the condition:

$$l \leq 2\sqrt{2rd - d^2} \tag{7.5.4}$$

where r is the local curvature radius. This loop involves the FFD transform computation at each step, to evaluate $\vec{P}_{n+1}$. This task is a huge time consuming. To remedy to this inconvenient, we use the polynomial coefficients instead, to calculate at each loop step the deformed points coordinates. The estimation of the polyline segment is straightforward:

7.5.2 Polyline Segment Estimation

In case of a deformed ray in the region $\mathbb{D}$, given the second degree polynomial coefficients, we apply the FFD transform to the start point t_s mentioned above to obtain the origin of the new polyline segment. Then, we have to look for the other extremity of the polyline segment. For that, we first define a maximum allowed error err_{max} as well as a maximum allowed length for a segment polyline Max_Seg_Length. Then, we suppose the segment length equal to Max_Seg_Length and evaluate the error between the considered part of the deformed ray represented by a curve and the polyline segment, by calculating the area in between as shown in figure 7.8. If this error is below the predefined maximum allowed error, then the polyline segment is equal to what we have supposed, otherwise we suppose the new polyline segment equal to the old one divided by two and so on until the error condition is satisfied and whenever the segment length is beyond a predefined minimum segment length Min_Seg_Length that we choose equal to half of the voxel unit. The approximation of the deformed ray by a polyline segment is described by the pseudo code given in algorithm 8.

After handling all the polyline segments in the current domain, we redo the same processing, considering the end point of the last domain as the start point of the new domain, and this until the end of the ray. All this process is described by the activity diagram illustrated by figure 7.9.

7.6 Implementation

Now that we can obtain a polyline segment, each segment of the polyline is still treated as a segment of a normal ray. But three modifications are made to enable

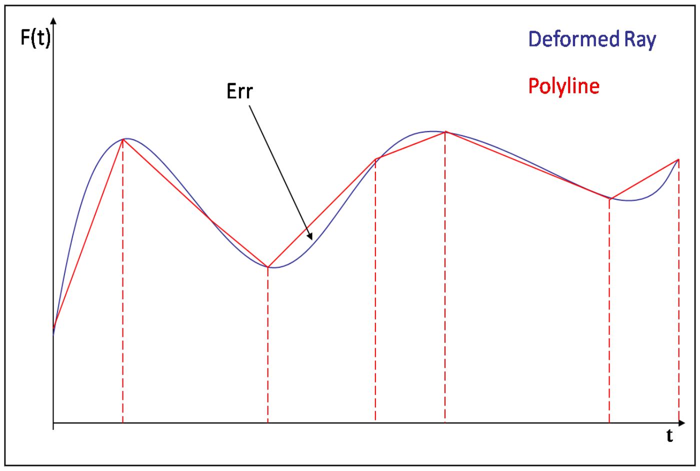


Figure 7.8: The process of transforming a deformed ray in polyline, given an error threshold

Algorithm 8 Get Polyline Segment: Pseudo-Code for Polyline Segment Evaluation. The inputs of this algorithm are the maximum allowed segment length and the undeformed ray *ray*. The output is the polyline segment *Poly_Seg* of length *seglength*.

```
VGLieee32 $Err$;
VGLieee32 $t$;
repeat
    $seglength* = 0.5$;
    $t = ray \to currentT[0] + seglength$;
    $Err = get\_error(t, ray \to currentT[0])$;
until $((Err > err\_max)$ and $(Seg\_Length > Min\_Seg\_Length)$ and $(t < poly\_domain \to end))$;
if $(t >= poly\_domain \to end)$ then
    $t = poly\_domain \to end$;
    VGLVector3f $r, p$;
    $calc\_poly(t, Poly\_Seg \to curr\_pt[1])$;
end if
$ray \to currentT[1] = t$;
$Poly\_Seg \to rayDirection = Poly\_Seg \to curr\_pt[1] - Poly\_Seg \to curr\_pt[0]$;
$Poly\_Seg \to rayStart = Poly\_Seg \to curr\_pt[0]$;
return $true$;
```

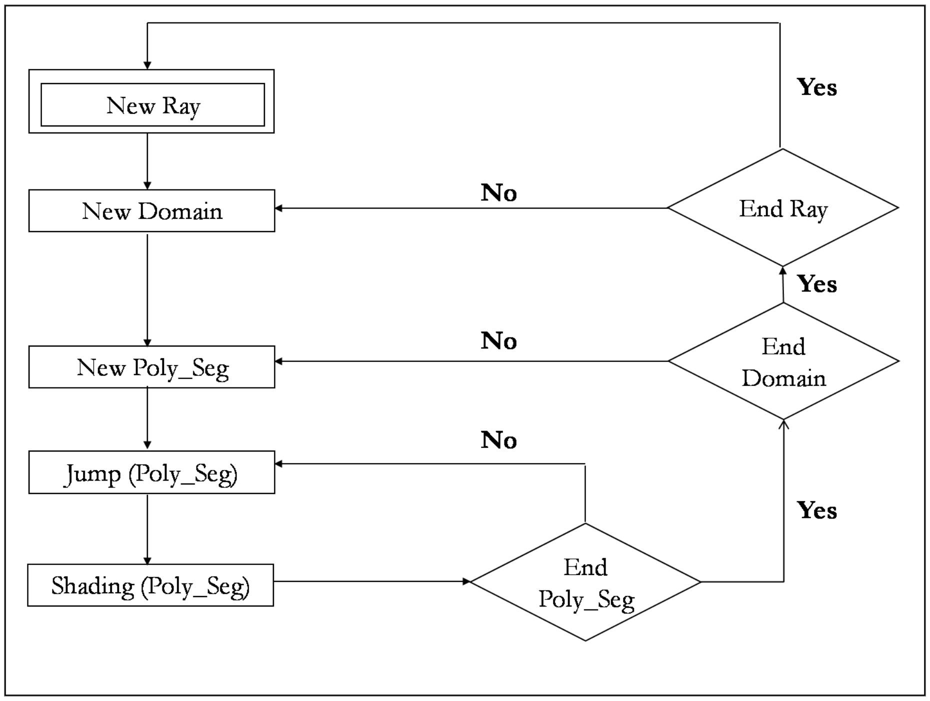


Figure 7.9: Ray Casting in deformed space

correct shading. One modification is for correct shading calculation, the two others for opacity compensation. After the volume is deformed, the intensity distribution within the deformed object is changed.

7.6.1 Opacity Calculation

The opacity per unit length along the deformed ray should be then compensated. The volume change, that a FFD imposes, is given by the Jacobian of the FFD [1, 53]. Let assuming that before deformation, the local absorption coefficient at a sample point is k, then the sample point opacity is given by:

$$\alpha_{orig} = 1 - e^{k.\Delta s}, \tag{7.6.1}$$

where Δs is the sampling space. Then, if we denote the determinant of the Jacobian matrix of the deformation function by Jac, the local absorption coefficient become k/Jac. Thus, the opacity of the deformed sample become:

$$\alpha_{def} = 1 - e^{k.\Delta s/Jac} = 1 - (1 - \alpha_{orig})^{1/Jac}, \tag{7.6.2}$$

The other opacity compensation is for the mismatch of the deformed polyline length to the standard sample unit. For this opacity correction, Chen [24] recalculates the opacity at the deformed sample point by evaluating the ratio between the actual sample interval and the standard sample space, in addition to the last modification. In our implementation, we avoid this additional computation, using the pre-integration lookup tables. In fact, we can get the opacity values by accessing the 2D lookup tables for all the possible sampling distance lengths as mentioned in chapter 5.

7.6.2 Shading

We now discuss shading in deformed space. We know that Phong shading model requires the normal vectors at each sample point, to be calculated. As for the opacity, when the object is deformed, the surface normal is also changed. Barr [1] derived that the normal vector transformation rule involves the inverse transpose of the Jacobian matrix, and can be expressed as follow:

$$\vec{n}(\vec{P'}) = Jac.(Jacobian^{-1})^T \vec{n}(\vec{P}) \tag{7.6.3}$$

where $\vec{n}(\vec{P})$ is the normal vector at the undeformed point $\vec{P}$ and $\vec{n}(\vec{P'})$ is the normal vector at the transformed position of the point $\vec{P}$. We evaluate the jacobian matrix, using the de Boor algorithm as described earlier in this chapter. Unlike Chen [24] who evaluated the jacobian matrix for each sample in the deformed space by deforming three extra points in the neighborhood of the sample point, in our deformation approach, since we assume that the deformation is linear for a given polyline segment, then instead of evaluating the jacobian matrix for each sample point, we do this only once for each polyline segment.

7.6.3 Algorithm's complexity

Over our program, two main functions contribute on the determination of the complexity of our algorithm. First, when creating a new domain via the *New Domain* function mentioned in figure 7.9, we invoke a function named Boor (see algorithm 7) which calculates the Tri-variate b-spline tensor product using the de Boor algorithm. The complexity of this function depends only on the B-spline degree p and is $O(p^4)$. In a second place, we also need for the shading correction, the evaluation of the jacobian matrix. This is done whenever a new polyline segment is created and consists on the calculation of the partial derivatives of the deformation function. Like for the previous function, its complexity function depends only on the b-spline degree p and is $O(p^4)$.

7.6.4 Optimization

As for the ray casting algorithm mentioned in chapter 6, the same acceleration methods can still be applied to the ray casting in deformed space. We cite the early ray termination, the space leaping, the coherence encoding. The only restriction is that due to limited length of the polyline segment, both space leaping and coherence encoding should be stopped whenever the end of the current polyline segment is reached.

Comparing to Chen [24] , we introduce to our implementation the pre-integration technique so that we can handle any sample interval length without problem.

7.7 Results

For our experiment, we use different volume data sets, namely the Engine ($128 \times 192 \times 256$ voxels), the Boston Teapot CT ($256^2 \times 178$ voxels) and the Engine CT ($256^2 \times 128$ voxels). All volume objects are $8bit$ data sets. We rendered the volume objects with different opacity transfer function, we choose the opacity transfer functions so that the volumes are opaque or semi-transparent. Images in figures 7.10 and 7.11 are examples of ray casting results of our algorithm.

As reported by Chen [24], the shading adjustment is very important for the correct display of the deformed shapes of volume objects. We therefore consider the shading adjustment for all our rendered images. In table 7.1, we give the rendering time needed for rendering deformed volumes. Comparing to the previous work of Chen [5], we achieve for the same volume size, better performance results. In fact, for example for a semi-transparent data set of size 67 Mb, we accomplish a speed up of factor $\simeq 1.99$. This is due to the use of the pre-integration technique for the opacity compensation, as well as the approximation of the deformed ray by an explicit polynomial which allows us to save the computation time due to the extra calculations needed to determine the normal vectors. These calculations consist on deforming three adjacent points for each sample point i.e. applying three times the FFD transform to calculate the normal transform matrix.

Nevertheless, the rendering time for deformed objects remains higher than the time needed for render undeformed objects. This is due to the extra calculations

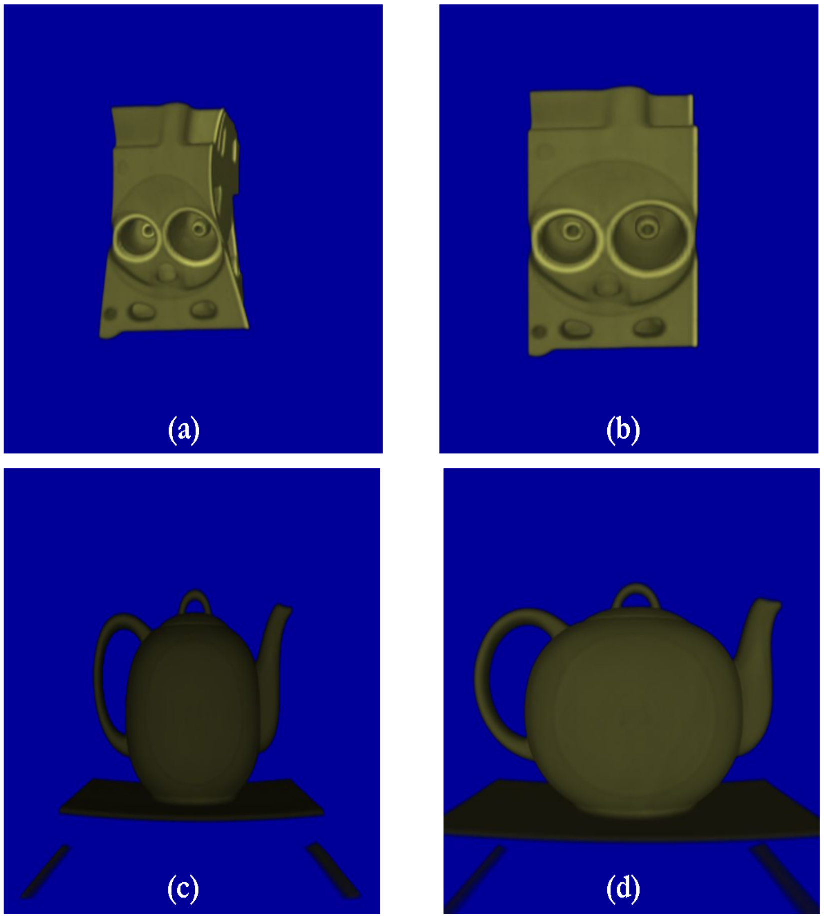
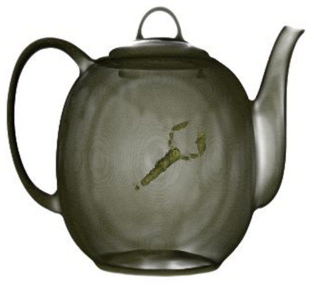
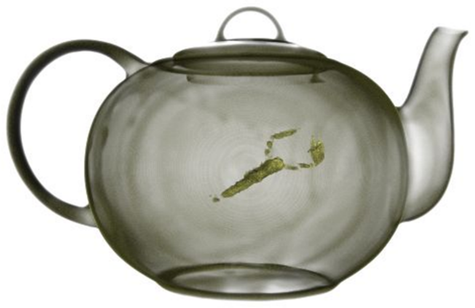


Figure 7.10: Rendering of deformed and undeformed opaque volume data sets: (a) The deformed engine. (b) The undeformed engine. (c) The deformed Boston Tea pot CT. (d) The undeformed Boston Tea pot CT.



(a)



(b)

Figure 7.11: Rendering of deformed and undeformed semi-transparent Boston Tea pot CT volume data set: (a) The deformed Boston Tea pot CT. (b) The undeformed Boston Tea pot CT.

Data set	Size (voxels)	Rendering time (s)
Engine (opaque)	$128 \times 192 \times 256$	9.413 ± 0.3
Engine (Semi-transparent)		11.89 ± 0.2
Boston Tea pot CT (opaque)	$256^2 \times 178$	10.45 ± 0.2
Boston Tea pot CT (Semi-transparent)		9.52 ± 0.1
Engine CT (opaque)	$256^2 \times 128$	12.41 ± 0.3
Engine CT (Semi-transparent)		19.25 ± 0.2

Table 7.1: Rendering time of deformed volume data sets

needed to determine the polyline segments. Even we use the acceleration techniques for ray casting, their effect remains limited because of the restricted length of the polyline segment.

7.8 Conclusion

In this chapter, we implemented a new method to render deformed objects. For that, we used the inverse ray deformation approach, it consists on deforming the ray casted into the volume data set and approximating it by a polyline segments which are rendered as normal rays. To accelerate our algorithm, we exploit coherency inside the volume and used pre-integrated technique for opacity compensation. We achieved a speed-up factor of up to 1.99 times, comparing to Chen[5] results. However, the computation time remains higher comparing to undeformed volume rendering case because of the restricted lengths of the polyline segments.

Chapter 8

Conclusion

Volume rendering is a technique which allows the visualization of volumetric data sets in order to extract information from them for further understanding and manipulation and this in different fields. The complexity of the transfer function, however can results in artifacts in image quality. In addition, the huge size of practical volume data sets or their semi-transparent aspect leads to a high computational volume rendering time.

To recapitulate from Chapter 1, the main objectives of this work were:

1. To design, specify and implement the pre-integrated volume rendering technique without simplification of the original rendering integral and this in first step for a linear gray value model and further for higher order polynomial one.

2. To implement and improve the existing coherency acceleration technique using the volume graphics library VGL for both shear-warp and ray casting volume rendering algorithms.

3. To develop advanced spline-based-ray-deformation approach which allows to

overcome the current limitations of the approach, i.e the excessive cost for computing the corrected image gradient by using the derivative of the spline-approximation.

In addition to these main objectives, a number of secondary objectives were decided upon. A summary of contributions made by this thesis and an evaluation of how well the initial objectives were met, now follows.

8.1 Summary of Contributions

A thorough background review of volume graphics was given in Chapter 2 as well as an introduction to the volume rendering technique which laid the groundwork for the subsequent volume rendering algorithms chapter. Chapter 3 then introduced the reader to the field of volume rendering and gave a detailed insight into the volume rendering algorithms we are interested in, and the related existing volume rendering acceleration methods. Further, chapter 4 presented the software development environment used in our implementation: the volume graphics library (VGL).

The first major objective was met in chapter 5 where a new pre-integrated volume rendering technique which extended the existing approach, has been developed. Our approach was based on the correct not simplified volume rendering integral, i.e. we considered that the normals depend on the opacity instead of the gray values. We also precompute the lookup tables for all Phong shading reflection model namely the ambient, diffuse and specular reflection. In addition, for our program, we consider different sampling interval distances for the lookup tables, so that it can be applied not only for ray casting algorithm but also to other volume rendering techniques where the sampling distance is not always constant, as for projected tetrahedra volume rendering algorithm, for coherence encoding or deformed objects. In our approach, we considered different gray value models: while previous work consider only a linear gray value model, we extended the existing pre-integrated volume rendering technique to quadratic and higher-order interpolation. For the linear case, we apply for the lookup table computation an incremental algorithm which allowed to calculate the lookup tables for different reflection model and different sampling distance lengths. We achieved a speed-up of factor of up to 10 times. We also compared the image quality results as well as the performance of our algorithm for different lookup table resolutions. We noticed that we can decrease the table sizes without affecting too much the image quality and thus minimizing the preprocessing time. This is more interesting when it is about the quadratic gray value model, since the time required for this case is higher than for the linear one, due to the dependency of the volume rendering integral in one addition parameter that we noticed by R. Thus, we achieved appealing image quality results for both interpolation models as well as an important speed up factor for the linear case. Then, we presented a new model which allows to determine the optimal LUT size for a given volume data set. However, our pre-integrated volume rendering for quadratic model has one limitation concerning the computation of the lookup tables for specular reflection. In fact due to the specular exponent present in the shading expression related to this part, we could not efficiently separate the halfway vector $\overrightarrow{H}$ from the volume rendering integral, thus increasing the lookup table parameters for specular reflection to four which drastically increase the computation time. Resolving this problem remains an open point.

The second major objective was met in chapter 6 where we implement and improve the existing coherence encoding approach. We first extended the coherence encoding technique to the shear-warp algorithm developed by Chen [5] by considering two voxel scanlines simultaneously when marching through the volume. In a second part, we applied coherence encoding to the ray casting algorithm. We first implement the Taylor Expansion based Extended Distance Coding (EDC) on the Volume Graphics Library, to calculate the coherence distance, view independently, for each voxel on the volume. Then we compare two algorithms that we have developed, one is using coherence encoding for volume rendering, the other combines both coherence encoding and pre-integrated technique in order to handle different coherent interval lengths. We noticed that for semi-transparent objects, we can achieve a speedup of factor up to 3. We achieved good images quality for both opaque and semi-transparent volumes. We also showed the influence of the encoding error on the image quality as well as on the algorithm performance. We find that whenever the error threshold increases, the image quality decreases but smoothly while the number of saved samples increase which explain the faster volume rendering time.

The third and last major objective of this thesis was met in chapter 7 where we developed more the ray-casting algorithm for deformed objects. We first gave an overview about the related work in this field and presented the mathematical background of the used techniques. Then, we used the inverse ray deformation technique to deform volumetric objects. We apply this technique to the ray-casting algorithm by bending the casted rays into the volume, instead of reconstructing the intermediate deformed object, thus avoiding the computational cost due to this operation and saving the extra memory which would be used to store the intermediate deformed volume. We used the free form deformation technique based on the uniform B-spline basis functions. In Chen's implementation, the viewing rays are adaptively divided to match the local deformation amplitude. This involves the calculation of the local curvature which requires many FFD transform computations when calculating the next position on the deformed ray. This task is time consuming. To overcome this problem, we use the property that the B-spline curve is a piecewise curve with each component a curve of degree p of the B-splines. Thus, we approximate each piecewise quadratic curve of the deformed ray by a polynomial by minimizing the distance in between. Unlike Chen, we explicitly apply the polynomial coefficients to get the next position on the deformed ray, saving the computations cost due to this operation. We also developed a shading calculation approach in the deformed space. We implement the true Phong shading in our method by backward transforming the normal vectors into the original volume space. This operation requires the Jacobian matrix evaluation. In his deformation approach, Chen [24] evaluated the Jacobian matrix for each sample in the deformed space by deforming three extra points in the neighborhood of the sample point. Unlike him, we calculate the Jacobian matrix only once for each polyline segment, since we assume the deformation being linear within a polyline segment. In addition, we compensate the opacity for the mismatch of the deformed polyline length to the standard sample unit, by using the pre-integration lookup tables to handle different segment lengths. Our experimental results showed that we can provide an additional speedup factor of 1.99, comparing to Chen.

8.2 Future Work

Further developments could be investigated in this work. First, to accelerate the pre-integrated lookup tables generation for the quadratic gray values model and to establish a suitable model to describe the specular reflection in this case. Second, the implementation of the coherency acceleration on consumer graphics cards. Consumer graphics cards allow to render objects faster than a CPU, due to the high internal memory bandwidth and processing power. The graphics cards would have the advantage that both CPU and graphics accelerator could work in parallel on a unified memory architecture for data exchange persuading many optimization options. Finally, in our deformation approach, the control points are moved by space function. This could be extended to an interactively deformation process.

Appendix A

Algorithms

Algorithm 9 Get opacity quad: The pseudo code for the quadratic opacity lookup table generation's algorithm. The input of this algorithm is the opacity transfer function

```
VGLieee32 $res$;
VGLieee32 $t = 0.0$;
VGLieee32 $alpha$;
VGLieee32 $d$;
VGLieee32 $h$;
VGLieee32 $inf$;
VGLieee32 $sup$;
VGLieee32 $samplingdist$;
VGLieee32 $M$;
VGLindex $f, b$;
VGLieee32 $rvalue$;
```

```
for $i = 0$ to $DIS$ do
  $samplingdist = P2[i - (DIS - VAR\_POW)]$;
  for $r = 0$ to $R\_FIELD$ do
    $d = samplingdist \times P2[4 - r]$;
    for $sf = 0$ to $N$ do
      for $sb = sf$ to $N$ do
        $f = LUT\_STEP * sf$;
        $b = LUT\_STEP * sb$;
        $alpha = (b - f)/d - d$;
        $M = f - (0.25 * alpha * alpha)$;
        $res = 0.0$;
        if $IsZero(alpha)$ then
          {In this case the integral is not defined on sf}
          $inf = f + IN\_INT\_STEP$;
          $sup = f + 1.0$;
          for $kk = 1$ to $INT\_STEP$ do
            $t = inf + kk * IN\_INT\_STEP$;
            $res+ = opt(M, t, material)$;
          end for
          $res+ = 0.5 * (opt(M, inf, material) + opt(M, sup, material))$;
          $res* = IN\_INT\_STEP$;
          for $i = f + 1$ to $b$ do
            $res+ = att\_quad(M, i, material)$;
          end for
```

```
else
  {$alpha \neq 0$}
  {We have two case according to the value of alpha:}
  {case 2 :The quadratic function has a minimum, so we have to split the
  interval of integration into two parts, because the quadratic function on
  each subinterval admit a different discriminant.}
  if $alpha < 0.0$ then
    VGLieee32$fb = M$;
    VGLindex$fbindex$;
    if $fb < 0.0$ then
      {Computation of the part from zero to f:}
      for $i = 0$ to $f$ do
        $res+ = att\_quad(M, i, material)$;
      end for
      {Computation of the part from zero to b:}
      for $i = 0$ to $b$ do
        $res+ = att\_quad(M, i, material)$;
      end for
    else
      $fbindex =$ (VGLindex)$ceil(fb)$;
      {Computation of the fractional part:}
      if $!IsZero(fbindex - fb)$ then
        if $abs(fbindex - fb) >= \_EPS$ then
          $h = (fbindex - fb) * IN\_INT\_STEP$;
          $inf = fb + h;$
          $sup = fbindex;$
          **for** $kk = 1$ to $INT\_STEP$ **do**
            $t = inf + kk * h;$
            $res+ = opt(M, t, material);$
          **end for**
          $res+ = 0.5 * (opt(M, inf, material) + opt(M, sup, material));$
          $res* = h;$
        **end if**
        {Computation of the part from fbindex to f:}
        **for** $i = fbindex$ to $f$ **do**
          $res+ = att\_quad(M, i, material);$
        **end for**
        $res* = 2.0;$
        {Computation of the part from f to b:}
        **for** $i = f$ to $b$ **do**
          $res+ = att\_quad(M, i, material);$
        **end for**
      **else**
        **if** $abs(fbindex - fb) >= \_EPS$ **then**
          $inf = inf = fb + IN\_INT\_STEP;$
          $sup = fbindex + 1.0;$
          $h = (sup - inf)/(INT\_STEP - 1);$
          **for** $kk = 1$ to $INT\_STEP$ **do**
```

```
              $t = inf + kk * h$;
              $res+ = opt(M, t, material)$;
            end for
            $res+ = 0.5 * (opt(M, inf, material) + opt(M, sup, material))$;
            $res* = h$;
          end if
        end if
        for $i = fbindex + 1$ to $f$ do
          $res+ = att\_quad(M, i, material)$;
        end for
        $res* = 2.0$;
        {Computation of the part from f to b:}
        for $i = f$ to $b$ do
          $res+ = att\_quad(M, i, material)$;
        end for
      end if
    else
      {case 1 : The quadratic function is monotonous and has a unique
      discriminant}
      for $i = f$ to $b$ do
        $res+ = att\_quad(M, i, material)$;
      end for
    end if
  end if
```

```
            VGLieee32 $sol = P2[r-4] * res$;
            $OPAC\_TAB[i][r][sf][sb] = OPAC\_TAB[i][r][sb][sf] = 1.0 - exp(-sol)$;
         end for
      end for
   end for
end for
return $true$;
```

Algorithm 10 The pseudo-code for the att_quad algorithm: The inputs of this algorithm are the gray value i, a constant M already calculated and the opacity transfer function stored in *material*. The output is the opacity value at the gray value i

```
VGLieee32 $res$;
$res = opt(M, i, material)$;
$res+ = 4.0 * opt(M, i + 0.5, material)$;
$res+ = opt(M, i + 1.0, material)$;
$res* = h6$;
return $res$;
```

Algorithm 11 Pseudo code for the opt algorithm: The inputs of this algorithm are the gray value t, a constant M already calculated and the opacity transfer function stored in *material*. The output is the opacity value at the gray value t

```
VGLieee32 $res$;
VGLieee32 $f0$;
VGLieee32 $f1$;
VGLieee32 $f2$;
VGLindex $i$;
VGLieee32 $tn$;
$tn = t + 0.5$;
$i =$ (VGLindex)$(tn)$;
**if** $i = 0$ **then**
   $f0 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $1$);
   $f1 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $0$);
   $f2 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $1$);
**else if** $i = N - 1$ **then**
   $f0 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $N - 2$);
   $f1 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $N - 1$);
   $f2 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $N - 2$);
**else**
   $f0 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $i - 1$);
   $f1 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $i$);
   $f2 = material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $i + 1$);
**end if**
$tn = tn - i$;
$res = 0.5 * (f2 + f0) - f1$;
$res *= tn * tn$;
$res += (tn * (f1 - f0))$;
$res += (0.5 * (f1 + f0))$;
$res /= 2.0 * sqrt(t - M)$;
**return** $res$;
```

Algorithm 12 Get opacity(`VGLSampleMaterial`$^{*}material, transfert^{*}tfo$): the opacity lookup table for linear gray value model. The inputs are the opacity transfer function. The output is the opacity LUT.

```
VGLieee32 $alpha$;
VGLieee32 $tmp$;
VGLieee32 $fact$;
VGLindex $sf, sb$;
$int\ j = 0$;
VGLieee32 $d = 2^{j-(DIS-VAR\_POW)}$; {d is the sampling distance}
for $j = 0$ to $DIS$ do
  for $b = 0$ to $N$ do
    for $f = 0$ to $b$ do
      $tmp\ =\ 0.0$;
      $alpha = 0.0$;
      $sf = LUT\_STEP * f$;
      $sb = LUT\_STEP * b$;
      if $f = b$ then
        $tmp = -d * material \rightarrow$ getLUTItem(VGL_MATERIAL_OPACITY, $sf$);
        $alpha = 1.0 - exp(tmp)$;
        $opacity[j][f][b] = alpha$;
      else
        $tmp = tfo \rightarrow att[sf] - tfo \rightarrow att[sb]$;
        $fact = d/(sb - sf)$;
        $tmp* = fact$;
        $alpha = 1.0 - exp(tmp)$;
        $opacity[j][f][b] = opacity[j][b][f] = alpha$;
      end if
    end for
  end for
  $d* = 2.0$;
end for
```

Bibliography

[1] A.H. Barr, *Global and local deformations of solid primitives*, Proceedings of the 11th annual conference on Computer graphics and interactive techniques (1984), 21–30.

[2] GG Cameron and PE Undrill, *Rendering volumetric medical image data on a SIMD-architecture computer*, Proceedings of the Third Eurographics Workshop on Rendering (1992), 135–145.

[3] K.M. Case and P.F. Zweifel, *Linear transport theory*, Addison-Wesley Pub. Co Reading, Mass, 1967.

[4] S.S. Chandrasekhar, *Radiative Transfer*, Courier Dover Publications, 1960.

[5] H. Chen, *Fast volume rendering and deformation algorithms*, Ph.D. thesis, Department of Mathematics and Computer Science, University of Mannheim, Germany, September 2001.

[6] H. Chen, J. Hesser, B. Vettermann, and R. Männer, *An Adaptive Distance-coding Based Volume Rendering Accelerator*, Proceedings of the 1st International Game Technology Conference, Hongkong (2001).

[7] Y. Chen, Q.H. Zhu, A. Kaufman, and S. Muraki, *Physically-based animation of volumetric objects*, Computer Animation 98. Proceedings (1998), 154–160.

[8] J. Danskin and P. Hanrahan, *Fast algorithms for volume ray tracing*, Proceedings of the 1992 workshop on Volume visualization (1992), 91–98.

[9] K. Engel, M. Kraus, and T. Ertl, *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (2001), 9–16.

[10] K. Engel, M. Kraus, and T. Ertl, *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*, 2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware (New York, NY, USA), ACM Press, 2001, pp. 9 – 16.

[11] S. Fang and R. Srinivasan, *Volumetric CSG–A Model-Based Volume Visualization Approach*, Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualisation (1998), 88–95.

[12] S. Fang, R. Srinivasan, R. Raghavan, and J.T. Richtsmeier, *Volume morphing and renderingAn integrated approach*, Computer Aided Geometric Design **17** (2000), no. 1, 59–81.

[13] T. Foley and J. Sugerman, *KD-tree acceleration structures for a GPU raytracer*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (2005), 15–22.

[14] J. Freund and K. Sloan, *Accelerated volume rendering using homogeneous region encoding*, Proceedings of the 8th conference on Visualization'97 (1997).

[15] H. Geyer, A. Seyfarth, and R. Blickhan, *Spring-mass running: simple approximate solution and application to gait stability*, J. Theor. Biol **232** (2005), no. 3, 315–328.

[16] S. Gibson and B. Mirtich, *A survey of deformable modeling in computer graphics*, MERL, TR-97 **19** (1997).

[17] AS Glassner, *Space subdivision for fast ray tracing*, Tutorial: computer graphics; image synthesis table of contents (1988), 160–167.

[18] A.S. Glassner, *An Introduction to Ray Tracing*, Morgan Kaufmann, 1989.

[19] V. Goel and A. Mukherjee, *An optimal parallel algorithm for volume ray casting*, The Visual Computer **12** (1996), no. 1, 26–39.

[20] M. Grigoriu, *Stochastic Calculus: Applications in Science and Engineering*, Birkhauser, 2002.

[21] Meister Eduard Gröller and Werner Purgathofer, *Coherence in computer graphics*, Tech. Report TR-186-2-95-04, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 1995.

[22] T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, HP Meinzer, and HJ Baur, *VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine*, development **6** (1995), no. 7, 8–9.

[23] V. Havran and J. Bittner, *LCTS: Ray Shooting using Longest Common Traversal Sequences*, Computer Graphics Forum **19** (2000), no. 3, 59–70.

[24] H.Chen, J.Hesser, and R.Mnner, *Ray casting free-form deformed-volume objects*, The Journal of Visualization and Computer Animation **14** (2003), 61–72.

[25] T. He, S. Wang, and A. Kaufman, *Wavelet-based volume morphing*, Visualization, 1994., Visualization'94, Proceedings., IEEE Conference on (1994), 85–92.

[26] H.C. Hege, T. Höllerer, and D. Stalling, *Volume Rendering: Mathematical Models and Algorithmic aspects*, ZIB, 1994.

[27] J. Hesser, *The VIRIM Project: Design and Realization of a Real Time Direct Volume Rendering System for Medical Applications*, VDI-Verl, 2000.

[28] J.F. Hughes, *Scheduled Fourier volume morphing*, Proceedings of the 19th annual conference on Computer graphics and interactive techniques (1992), 43–46.

[29] S.U. Jo and C.S. Jeong, *A Parallel Volume Visualization Using Extended Space Leaping Method*, Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia (2000), 296–305.

[30] J.T. Kajiya, *The rendering equation*, International Conference on Computer Graphics and Interactive Techniques, 1986, pp. 143–150.

[31] Martin Kraus, Wei Qiao, and David S. Ebert, *Projecting tetrahedra without rendering artifacts*, VIS '04: Proceedings of the conference on Visualization '04 (Washington, DC, USA), IEEE Computer Society, 2004, pp. 27–34.

[32] W. Krueger, *The application of transport theory to visualization of 3d scalar data fields*, Proc. IEEE Visualization (1990), 273–280.

[33] Y. Kurzion and R. Yagel, *Space deformation using ray deflectors*, Proc. the 6th Eurographics Workshop on Rendering (1995), 21–32.

[34] P. Lacroute and M. Levoy, *Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform*, Proc. SIGGRAPH, 1994, pp. 451–458.

[35] M. Levoy, *Display of surfaces from volume data*, Proc. IEEE Computer Graphics and Applications **8** (1988), no. 3, 29–37.

[36] M. Levoy, *Efficient ray tracing of volume data*, ACM Transactions on Graphics (TOG) **9** (1990), no. 3, 245–261.

[37] M. Levoy, *Volume rendering by adaptive refinement*, The Visual Computer **6** (1990), no. 1, 2–7.

[38] E. Lum, B. Wilson, and K. Ma, *High-quality lighting and efficient pre-integration for volume rendering*, Proceedings of the Joint Eurographics-IEEE TVCG Symposium on Visualization 2004, 2004.

[39] N. Max, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics **1** (1995), no. 2, 99–108.

[40] N. Max, P. Crawfis, and P. Hanrahan, *Area and volume coherence for efficient visualization of 3d scalar functions*, Proc. IEEE Symposium on Volume Visualization, vol. 24, 1990, pp. 27–33.

[41] K. Moreland and E. Angel, *A fast high accuracy volume renderer for unstructured data.*, VolVis, 2004, pp. 9–16.

[42] J. Neider, T. Davis, and M. Woo, *OpenGL. Programming guide*, Addison-Wesley Reading, Mass, 1997.

[43] A. Pearce and D. Jevans, *Exploiting shadow coherence in ray tracing*, Graphics Interface 91 (1991), 109–116.

[44] B.T. Phong, *Illumination for computer generated pictures*, Communications of the ACM **18** (1975), no. 6, 311–317.

[45] X. Provot, *Deformation constraints in a mass-spring model to describe rigid cloth behavior*, Graphics Interface **95** (1995), 147–154.

[46] A. Reshetov, A. Soupikov, and J. Hurley, *Multi-level ray tracing algorithm*, Proceedings of ACM SIGGRAPH 2005 **24** (2005), no. 3, 1176–1185.

[47] S. Roettger and T. Ertl, *A two-step approach for interactive pre-integrated volume rendering of unstructured grids*, Proceedings of the 2002 IEEE symposium on Volume visualization and graphics (Piscataway, NJ, USA), IEEE Press, 2002, pp. 23 – 28.

[48] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, *Smart hardware-accelerated volume rendering*, Proceedings of the symposium on Data visualisation 2003 (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2003, pp. 231 – 238.

[49] S. Roettger, M. Kraus, and T. Ertl, *Hardware-accelerated volume and isosurface rendering based on cell-projection*, Proc. IEEE Vis 2000 (Washington, DC, USA), IEEE Computer Society, 2000, pp. 109 – 116.

[50] P. Sabella, *A rendering algorithm for visualizing 3D scalar fields*, International Conference on Computer Graphics and Interactive Techniques, vol. 22, 1988, pp. 51–58.

[51] H. Samet and RE Webber, *Hierarchical data structures and algorithms for computer graphics. I. Fundamentals*, Computer Graphics and Applications, IEEE **8** (1988), no. 3, 48–68.

[52] J. P. Schulze, M. Kraus, U. Lang, and T. Ertl, *Integrating pre-integration into the shear-warp algorithm*, Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics (New York, NY, USA), ACM Press, 2003, pp. 109 – 118.

[53] Thomas W. Sederberg and Scott R. Parry, *Free-form deformation of solid geometric models*, SIGGRAPH Comput. Graph. **20** (1986), no. 4, 151–160.

[54] A. Sheffer and V. Kraevoy, *Pyramid coordinates for morphing and deformation*, 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on (2004), 68–75.

[55] KR Subramanian and DS Fussell, *Applying space subdivision techniques to volume rendering*, Visualization, 1990. Visualization'90., Proceedings of the First IEEE Conference on (1990), 150–159.

[56] J. Sweeney and K. Mueller, *Shear-warp deluxe: The shear-warp algorithm revisited*, Proc. of the symposium on Data Visualisation, 2002, pp. 95–104.

[57] B. Vettermann, J. Hesser, R. Männer, H. Singpiel, and A. Kugel, *Implementation of Algorithmically Optimized Volume Rendering on FPGA-Hardware*, IEEE Visualization (1999), 13–16.

[58] I. Wald, T. Ize, A. Kensler, A. Knoll, and S.G. Parker, *Ray tracing animated scenes using coherent grid traversal*, ACM Transactions on Graphics (TOG) **25** (2006), no. 3, 485–493.

[59] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, *Interactive Rendering with Coherent Ray Tracing*, Computer Graphics Forum **20** (2001), no. 3, 153–165.

[60] T. Whitted, *An improved illumination model for shaded display*, International Conference on Computer Graphics and Interactive Techniques (2005).

[61] P. Williams and N. Max, *A volume density optical model*, Proceedings of the 1992 workshop on Volume visualization (New York, NY, USA), ACM Press, 1992, pp. 61 – 68.

[62] P.L. Williams, N.L. Max, and C.M. Stein, *A high accuracy volume renderer for unstructured data*, IEEE Transactions on Visualization and Computer Graphics **4** (1998), no. 1, 37–54.

[63] R. Yagel and Z. Shi, *Accelerating volume animation by space-leaping*, Visualization, 1993. Visualization'93, Proceedings., IEEE Conference on (1993), 62–69.

[64] K.J. Zuiderveld, A.H.J. Koning, and M.A. Viergever, *Acceleration of ray-casting using 3D distance transforms*, Proceedings of Visualization in Biomedical Computing **1808** (1992), 324–335.