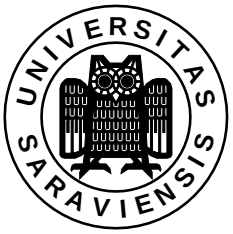
Interactive Volume Ray Tracing

Gerd Marmitt

Computer Graphics Group
Saarland University
Saarbrücken, Germany

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes



Betreuender Hochschullehrer / Supervisor:

Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Gutachter / Reviewers:

Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Prof. Dr. rer. nat. Elmar Schömer, Johannes Gutenberg Universität, Mainz, Germany

Dekan / Dean:

Prof. Dr. rer. nat. Joachim Weickert

Eingereicht am / Thesis submitted:

26. Juni 2008 / June 26th 2008

Datum des Kolloquiums / Date of defense:

18. Dezember 2008 / December 18th 2008

Prüfungskommission / Committee:

Prof. Dr. rer. nat. Sebastian Hack, Universität des Saarlandes
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes
Prof. Dr. rer. nat. Elmar Schömer, Johannes Gutenberg Universität
Dr. Andreas Hildebrandt, Universität des Saarlandes

Gerd Marmitt
Lehrstuhl für Computergraphik, Campus E 1 1
Universität des Saarlandes
66123 Saarbrücken
marmitt@graphics.cs.uni-sb.de

Abstract

Volume rendering is one of the most demanding and interesting topics among scientific visualization. Applications include medical examinations, simulation of physical processes, and visual art. Most of these applications demand interactivity with respect to the viewing and visualization parameters. The ray tracing algorithm, although inherently simulating light interaction with participating media, was always considered too slow. Instead, most researchers followed object-order algorithms better suited for graphics adapters, although such approaches often suffer either from low quality or lack of flexibility.

Another alternative is to speed up the ray tracing algorithm to make it competitive for volumetric visualization tasks. Since the advent of modern graphic adapters, research in this area had somehow ceased, although some limitations of GPUs, e.g. limited graphics board memory and tedious programming model, are still a problem. The two methods discussed in this thesis are therefore purely software-based since it is believed that software implementations allow for a far better optimization process before porting algorithms to hardware. The first method is called *implicit kd-tree*, which is a hierarchical spatial acceleration structure originally developed for iso-surface rendering of regular data sets that now supports semi-transparent rendering, time-dependent data visualization, and is even used in non volume-rendering applications. The second algorithm uses so-called *Plücker coordinates*, providing a fast incremental traversal for data sets consisting of tetrahedral or hexahedral primitives. Both algorithms are highly optimized to support interactive rendering of volumetric data sets and are therefore major contributions towards a flexible and interactive volume ray tracing framework.

Kurzfassung

Die Visualisierung von volumetrischen Daten ist eine der interessantesten, aber sicherlich auch schwierigsten Anwendungsgebiete innerhalb der wissenschaftlichen Visualisierung. Im Gegensatz zu Oberflächenmodellen, repräsentieren solche Daten ein semi-transparentes Medium in einem 3D-Feld. Anwendungen reichen von medizinischen Untersuchungen, Simulation physikalischer Prozesse bis hin zur visuellen Kunst. Viele dieser Anwendungen verlangen Interaktivität hinsichtlich Darstellungs- und Visualisierungsparameter. Der Ray-Tracing- (Stahlverfolgungs-) Algorithmus wurde dabei, obwohl er inhärent die Interaktion mit einem solchen Medium simulieren kann, immer als zu langsam angesehen. Die meisten Forscher konzentrierten sich vielmehr auf Rasterisierungsansätze, da diese besser für Grafikkarten geeignet sind. Dabei leiden diese Ansätze entweder unter einer ungenügenden Qualität respektive Flexibilität.

Die andere Alternative besteht darin, den Ray-Tracing-Algorithmus so zu beschleunigen, dass er sinnvoll für Visualisierungsanwendungen benutzt werden kann. Seit der Verfügbarkeit moderner Grafikkarten hat die Forschung auf diesem Gebiet nachgelassen, obwohl selbst moderne GPUs immer noch Limitierungen, wie beispielsweise der begrenzte Grafikkartenspeicher oder das umständliche Programmiermodell, enthalten. Die beiden in dieser Arbeit vorgestellten Methoden sind deshalb vollständig softwarebasiert, da es sinnvoller erscheint, möglichst viele Optimierungen in Software zu realisieren, bevor eine Portierung auf Hardware erfolgt. Die erste Methode wird *impliziter Kd-Baum* genannt, eine hierarchische und räumliche Beschleunigungstruktur, die ursprünglich für die Generierung von Isoflächen reguläre Gitterdatensätze entwickelt wurde. In der Zwischenzeit unterstützt sie auch die semi-transparente Darstellung, die Darstellung von zeitabhängigen Datensätzen und wurde erfolgreich für andere Anwendungen eingesetzt. Der zweite Algorithmus benutzt so genannte *Plücker-Koordinaten*, welche die Implementierung eines schnellen inkrementellen Traversierers für Datensätze erlauben, deren Primitive Tetraeder beziehungsweise Hexaeder sind. Beide Algorithmen wurden wesentlich optimiert, um eine interaktive Bildgenerierung volumetrischer Daten zu ermöglichen und stellen deshalb einen wichtigen Beitrag hin zu einem flexiblen und interaktiven Volumen-Ray-Tracing-System dar.

for Angela

Acknowledgments

It is unquestionable that many students, colleagues, relatives, and friends have contributed significantly in many ways to make this thesis finally happen. During my work in the Computer Graphics Group at the Saarland University, I advised Jens-Michael Weber, Andreas Kleer, Heiko Friedrich, Javor Kalojanov, and Roman Brauchle in their Bachelor or Master Theses. Most of their work was used in several publications within the last four years and became therefore also part of this thesis. Besides programming essential parts of the entire framework, discussions were fruitful for both sides.

The same can be said about my former colleagues Andreas Dietrich, Heiko Friedrich (again), Andreas Pomi, Jörg Schmittler, and Sven Woop, who helped me understand many issues in the computer graphics area. A special thanks goes to Heiko Friedrich since most research projects were realized with his help.

Devoting years to education is in my point of view extremely difficult without having relatives and family for support. Most support came from my parents and three sisters, but I also would like to thank my friends Tanja Warken, Matthias Buchmann, Lars Baldes, Marc Schmidt, and Carsten Spyra. Special thanks goes to Benedikt Fries, since he gave me the chance to see Japan right before I started writing this thesis.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Im Lichte bereits erlangter Erkenntnis erscheint das glücklich Erreichte fast wie selbstverständlich, und jeder intelligente Student erfaßt es ohne zu große Mühe. Aber das ahnungsvolle, Jahre währende Suchen im Dunkeln mit seiner gespannten Sehnsucht, seiner Abwechslung von Zuversicht und Ermattung und seinem endlichen Durchbrechen zur Wahrheit, das kennt nur, wer es selbst erlebt hat.

Albert Einstein

Scientific visualization is an important research areas within the computer graphics community and covers a large variety of methods and applications. However, such applications usually demand processing and inspection of participating media. Rendering participating media is called volume graphics or volume rendering and is one of the most interesting (and difficult) topics, since it provides a detailed exploration of material in the context of physical, medical, and biological research.

Volumes are generally described as participating media in a 3D field, which takes light interaction into account. Such a 3D field may represent density, temperature, pressure, higher dimensional data like acceleration and velocity vectors, or even a combination of these properties. The remainder of this thesis will, however, be restricted to scalar fields since multi-dimensional fields are beyond the scope of this work.

Sources for 3D scalar fields are either measurement devices or computer simulations. Typical devices used for this task include computer tomography (CT) and magnetic resonance imaging (MRI), which are familiar to the general public. For example, a CT scanner produce images similar to x-rays from various orientations offering doctors a better diagnostic tool for their patients. Until now, physician often relied on 2D images, since 3D imaging tools were either too costly or lacked display quality. There can be, however, no doubt that 3D imaging offers far better examination capabilities and analysis which will lead to a growing demand in the near future.

Another field of application lies in simulation of computational fluid dynamics (CFD) or finite element methods (FEM). Both applications require 3D scalar fields for a proper representation of physical processes, e.g. temperature and pressure distribution in a combustion chamber over time. Usually, supercomputers are employed for such simulations since the underlying physical model requires complex calculations. Scientific visualization aids in the understanding of these physical processes.

A third community which has started using volume rendering very recently is visual artists. More and more artists are impressed by the expressiveness and possibilities of volume rendering. In computer games and other areas of visual art, volumetric rendering effects can be used for describing

non-solid objects, e.g. fluids, gases, and natural phenomena like fog, clouds, and fire. Volume rendering is used here to add visual clues to virtual realities as a supplement to traditional surface models.

Computer games rely completely on graphics boards for rendering, which do not directly support rendering of volumetric data. Due to their limited flexibility, which only changed recently, most proposed algorithms are fast but restricted to the rasterization of surfaces. All primitives are projected onto the screen before applying visibility sorting. Volumetric primitives, however, require a sampling and sorting within the primitive for a proper visualization. Ray tracing naturally supports this, since it imitates the physical model of light transport directly and can therefore be used in a more general way, i.e. supporting a variety of visualization tasks with only minor modifications of the basic algorithm. Yet, due to the lack of ray tracing hardware, it was always considered as too slow for interactive purposes.

The *OpenRT* project [Wald02a, Dietrich03] showed that this is no longer true in the case of polygonal data. This ray tracing engine consists of an efficient combination of processor-specific command sets, i.e. SIMD, highly optimized acceleration structures and the ability to cluster several consumer PCs together for rendering the final image. The latter results directly from its image-based rendering approach, i.e. the main loop runs over the image pixels and not scene objects. This allows for nearly linear scalability with respect to computational power. Since all pixels can be computed independently, doubling the number of (equally powerful) processors halves the time for computing the same image. For the same reason, ray tracing can profit directly from multi-core processors.

Once the basic ray tracing algorithm is implemented, it is easy to add varying types of shading models, e.g. Lambertian [Gouraud71] and Phong [Phong75] shading. More complex ray generation and shooting allows for global illumination with soft shadows and caustics (e.g. [Jensen96]). Additionally, ray tracing handles any kind of primitive (triangles, quads, nurbs, etc.) along the ray and returns the hit position of the first opaque object.

The entire framework was restricted to surface models consisting of triangles and could hence not handle volumetric data. The main focus of this thesis is therefore to add this functionality to the *OpenRT* rendering system by exploring two interesting concepts recently proposed in the area of volume rendering: *implicit kd-trees* [Wald05, Marmitt05, Marmitt08] and *Plücker-based volume traversal* [Marmitt05, Marmitt06b, Marmitt08]. It is worth noting that the presented algorithms can be implemented in any ray tracing-based rendering system.

However, also practical issues are discussed. It is of major importance to distinguish first between different types of volumetric data, i.e. regular

and curvilinear grids as well as unstructured meshes. Since all data sets offer only a discrete signal, the reconstruction of a continuous signal by using interpolation is briefly discussed. The section continues with an overview of typical volume rendering techniques, i.e. iso-surface rendering, maximum-intensity projection, and semi-transparent rendering. It will be closed with a short overview of a generic volume rendering pipeline used by most volume renderers.

Chapter 3 describes different types of object-order approaches for volume rendering. They are considered as alternative rendering methods throughout this thesis. In essence, these alternative methods can be distinguished in *cell-projection*, *vertex-projection*, and *texture-mapping*. Their main advantage is rendering performance, since it is easier to implement them on graphics adapters. An exception is certainly the *shear-warp* algorithm, which is a fast software implementation for regular grids. It is not frequently used, however, due to its algorithm-inherent shortcomings, i.e. rendering artifacts from 45 degree viewpoints, memory consumption, etc. The chapter ends with a brief discussion of *custom hardware* implementations. Most, however, are not developed further due to their costly research and limited extensibility with respect to rapid advancements in that area.

Chapter 4 extensively discusses the first technique for allowing interactive volume ray tracing of regular grids. The *implicit kd-tree* is described and optimized for rendering iso-surfaces of static volume data sets first. Recent advancements enable a fast introspection even of massive volumetric data sets. As the following section shows, hierarchical semi-transparent rendering can be added with only minor modifications. Conclusions and future work close this chapter.

As Chapter 5 shows, the *implicit kd-tree* can also be used for rendering time-dependent data. The first alternative introduces a concurrent update mechanism using a shared-memory system and is therefore well-suited for small data sets. Larger data sets can be rendered by the newly developed 4D kd-tree, although this acceleration structure is not competitive in its current state. Two special sections cover the usage of the *implicit kd-tree* in non-volume rendering areas. In a first example, a terrain rendering system consisting of 90 trillion triangles is described where the implicit kd-tree determines which tiles are pierced by a ray from the elevation map. The entire system achieves near-interactive frame rates on a shared-memory system with 16 cores. The second example uses the idea of the *implicit kd-tree* to support dynamic rendering of polygonal data. This is made possible using the *bounded kd-tree*.

Chapter 6 describes an incremental traversal algorithm for curvilinear and unstructured data sets. In contrast to the previously discussed hierarchical

acceleration structure, this incremental traverser supports all volume rendering techniques with only minor modifications, i.e. semi-transparent rendering is supported directly. Care has to be taken to allow for an interactive performance since this is the main goal of all algorithms proposed here. This goal is made possible using the concept of *Plücker coordinates* and bilinear patches. While *Plücker coordinates* solely suffice to render unstructured, i.e. tetrahedralized, data efficiently, a hybrid approach of *Plücker coordinates* and bilinear patch intersections for curvilinear grids consisting of hexahedral primitives offers both a better quality and a higher performance. Of course this chapter also closes with results, drawn conclusions and future work.

Finally Chapter 7 briefly summarizes all achievements made towards an interactive and flexible volume ray tracing system. It also suggests how such a system might look and states crucial components.

Chapter 2
Ray Tracing & Volume Graphics

The rules of the game are laid down. We all have to play, buddy!

"Brazil"

This chapter covers main concepts of volume rendering as used throughout this thesis. It starts with Section 2.1, which provides an overview of the well-known ray tracing algorithm. As will later be shown, this global, image-space rendering approach has a variety of advantages compared to rasterization. While this fact is in general not questioned, the performance for rendering images with ray tracing is often cited as its largest drawback.

However, rendering polygonal data with ray tracing at interactive frame rates has been established for several years. Research groups in Utah (USA) and Saarbrücken (Germany) demonstrated interactive ray tracing for supercomputers *(*-ray)* [Parker99a] and a cluster of commodity PC's *(OpenRT)* [Wald02a, Wald04a]. The latter was more successful since a cluster of consumer PC's is less expensive compared to a high-end supercomputer, such as an *SGI Reality Monster*. Furthermore *OpenRT* was extended in a variety of ways, including global illumination [Wald02b, Günther04], video-textures for mixed-reality applications [Pomi03], and free-form surfaces [Benthin04]. As a pure software implementation, it can also be used for displaying highly complex models [Wald01b, Wald04b] such as, CAD data [Dietrich05b], or natural plant scenes [Dietrich05a, Dietrich06]. Additionally, the ray tracing algorithm was successfully ported to several hardware prototypes. Using FPGAs as a platform allows for an efficient development while still keeping close to custom chip design issues, i.e. ASICs. While the first implementation offered only a limited number of features [Schmittler02], the latest developments include freely programmable shaders [Woop05] as well as efficient handling of dynamic scenes [Woop06].

However, all of these implementations consider surface (polygonal) data only. Evaluating light transfer at surfaces only does not take into account interaction with an atmosphere or the interior of objects. In contrast, volume rendering describes a wide range of techniques for generating images from a 3D (scalar) field. It seems therefore only plausible to extend the existing interactive ray tracing system to handle volumetric data. The following section introduces the key components and advantages of a ray tracing system first, while Section 2.2 discusses the theoretical background and practical issues for handling volumetric data. It will be shown that ray tracing naturally supports volumetric data and allows for easy integration into the ray tracing system.

2.1 Ray Tracing

Ray tracing is a well-known technique for producing realistic images by simulating the transport of light [Appel68, Kay79, Whitted80]. In contrast to rasterization, the visible objects in a given scene are determined at the pixel level, i.e. this algorithm works in image-order. The final image is rendered pixel-wise by searching for the nearest visible object from the eye point. Pixel-wise processing makes the algorithm relatively slow compared to rasterization, especially for objects covering large regions in screen space. On the other hand, implementing a generic ray tracer is rather straightforward. Shadows and point light sources can be easily added, and for reflection and refraction, it is sufficient to implement the corresponding physical equations for ray shooting, e.g. Bears law, Snell's law, or Fresnel equation, into the system.

Algorithm 1 A Generic Ray Tracing Algorithm.

```
for p = 0..#Pixels do
  Compute viewing ray
  for o = 0..#Objects do
    if o is hit at ray parameter t and t < t_hit then
      hit := true
      o_hit := o
      t_hit := t
    end if
  end for
  if hit = true then
    Set pixel color to o_hit color
  else
    Set pixel color to background color
  end if
end for
```

As can be seen from Algorithm 1, a basic ray tracing engine consists of two nested loops running over all pixels (outer loop) and over all objects (inner loop).[1] The inner loop enumerates all objects, calculates the distance from the eye point for each object, and returns the object closest to the eye point.[2] The pixel color is either set to the color of the returned object or the background color, respectively, if no object was found.

[1] In fact, swapping inner and outer loops results in a basic rasterization algorithm.

[2] The visibility sorting applied here is known as the *painters algorithm* [de Berg00].

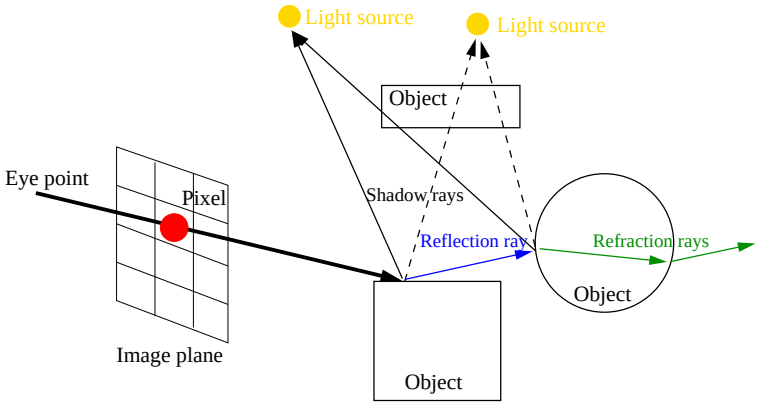


Figure 2.1: In ray tracing, rays are shot from the eye point through each pixel of the image plane. To account for shadow or material properties like reflection and refraction, secondary rays are traced recursively in the scene.

So far, this algorithm considers primary (or viewing) rays only, which is usually referred as to ray casting. To add more visual cues, it is necessary to continue shooting rays. For example, shooting rays from the intersection point to all light sources determine whether this part of an object lies in a shadow. This is achieved by checking whether another primitive is located between the light source and the object currently intersected. Reflection and refraction might also be added by applying the corresponding physical laws to the incoming ray and continue shooting. It is obvious that ray shooting can continue from that point until no further objects are intersected or the material properties of the intersected primitive do not request shooting of further rays. All subsequent rays are called secondary (or higher-order) rays. Figure 2.1 illustrates the ray tracing algorithm. In general, a ray is only traced until a pre-defined level of recursion, referred to as depth, is reached, since the contribution to the final pixel color diminishes with each level. Instead pre-defining the level of recursion, recursion may also stop if the contribution is too small.

Another extension towards more realistic images is to shoot more than one ray per ray depth. For example, image quality can be improved using super-sampling, i.e. shooting rays 'in-between' pixels according to a certain pattern, since this avoids aliasing (staircase) artifacts. More natural appearing soft shadows can be computed by implementing area light sources and sampling the light source area instead of a single boolean shadow test. Global illumination effects like color bleeding can be achieved by sampling the hemisphere from each intersection point with several rays and accumulating their

contributions. Of course, all these effects lead to an exponential growth of rays and reduces the rendering performance. The challenge is here to achieve a realistic looking image with a minimal number of rays.

Applying the basic principles to volume graphics, the problem is shifted one level deeper. At first, efficient algorithms for shooting rays through volumetric data need to be investigated before applying advanced illumination effects. Traversing a volume means computing the light interaction with a participating medium. Basic types of interaction are emission (i.e. increasing radiance), absorption (i.e. decreasing radiance), and scattering (i.e. radiance leaves in directions other than the light ray and therefore increases (i.e. out-scattering) and decreases (i.e. in-scattering) energy at the same time). This interaction must to be evaluated at all positions in the 3D volume. Gaseous materials are the most common example. Volume visualization is therefore more computationally demanding compared to polygonal rendering. The following section will therefore not only provide a theoretical background but will also discuss practical issues. As it will turn out, ray tracing provides an easily implemented solution for volume graphics.

2.2 Volume Graphics

For volume graphics, a participating medium needs to be modeled along with the actual light-transport mechanism. While photo-realistic rendering requires a physically accurate description of the participating medium, scientific visualization tasks aim at emphasizing certain information implicitly encoded in the medium.

In other words, the goal of direct volume visualization is the visual extraction of information from a 3D scalar field[3], which can be interpreted as a mapping:

$$\phi : \mathbb{R}^3 \rightarrow \mathbb{R},$$

i.e. from a 3D space to a single-component value. This 3D scalar field originates either from measurements (e.g. CT, MRT) or simulations (e.g. CFD, FEM). The following paragraphs discuss the fundamental equations for light transfer and follows closely in its development of the volume rendering integral [Engel06, Hege93]. The usage of the complete volume rendering integral is unfortunately only of limited use for interactive applications. It is therefore necessary to discuss a simplified and discrete version afterwards so that numerical methods can be used (see Section 2.2.2). Finally it is then possible

[3]Be reminded, that multi-dimensional fields are not a topic of this thesis.

to put all discussed parts together for a generic volume rendering pipeline (see Section 2.2.6).

2.2.1 Transport of Light

For this model it is assumed that light propagates along straight lines in space as long as it does not interact with a participating medium, i.e. relativistic effects are generally not considered. Typically, the following three types of interactions are distinguished [Engel06]:

Emission. The participating medium actively emits light, i.e. the radiated energy is increased. Hot gas converting heat into radiative energy is an example.

Absorption. Here, the light is absorbed by converting radiated energy into heat. Sun collectors converting sun light into hot water are an example.

Scattering. Essentially, scattering can be interpreted as changing the direction of light propagation. It can be further distinguished in *elastic scattering* (wavelength energy of photons is not a affected) and *inelastic scattering* (wavelength energy of photons changes).

Each of these three types of interaction affect the amount of radiated energy along a light ray. This light energy is usually described by its *radiance* L.[4] This term can be derived by combining *radiant power* ϕ, *irradiance* E, and the solid angle Ω:

$$\phi = \frac{dQ}{dt}\left[W = \frac{J}{s}\right] \qquad E = \frac{d\phi}{dA}\left[\frac{W}{m^2}\right], \tag{2.1}$$

which leads to the power arriving at or leaving from a surface per solid angle and per unit projected area, *radiance* L is defined as

$$L = \frac{dQ}{dA_{\perp} d\Omega dt}\left[\frac{W}{m^2 \cdot sr}\right]. \tag{2.2}$$

In other words, L is characterized as radiance energy Q per unit area A, solid angle Ω, and time t. $dA_{\perp} = dAcos\theta$ where θ denotes the angle between the surface normal and the incoming light ray. Hence, the subscript $\perp$ indicates that the area is measured as projected along the light direction.

[4] In [Engel06] the symbol I is used for radiance despite the fact that Glassner [Glassner95] uses L for radiance, since this would lead to confusion with the intensity term. The Glassner notation is used in the following.

Physically, *radiance* is stated in *watts per steradian per square metre*. The unit of the solid angle is dimensionless. The physical units will be omitted from here on to make the equations more readable.[5]

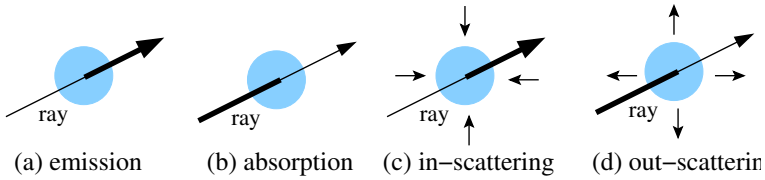


Figure 2.2: A light ray traversing a participating medium interacts with it in several ways. While emission increases the radiance for each ray (a), absorption has the opposite effect (b). Scattering may either increase the outgoing radiance (c), or decrease the outgoing radiance (d).

To begin, consider the effect of a medium on the radiative energy along a ray. Except for a vacuum, a medium generally affects the radiance energy in different ways. As illustrated in Figure 2.2, emission increases the light energy, while absorption reduces the light energy. Scattering effects are further distinguished in *in-scattering*, where additional energy is redirected into the ray direction, and *out-scattering*, where the outgoing ray energy is reduced since light is scattered in different directions. Note that this is independent from *elastic* and *inelastic scattering*. The latter concerns only changes of the underlying wavelength and is therefore not further considered here. Combining the effects for emission, absorption and scattering results in the following equation for light transfer:

$$\omega \cdot \nabla_x L(x, \omega) = -\chi L(x.\omega) + \eta. \tag{2.3}$$

The term $\omega \cdot \nabla_x L$ denotes the dot product between light direction ω and the gradient $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ of the radiance L at position x, i.e. the directional derivative taken along the light direction. χ represents the total absorption coefficient defining the rate of light attenuation with respect to the medium and is therefore dependent on the radiance energy at point x. A second term, η needs to be added representing the emission (i.e. describing the extent to which radiative energy is increased through the participating medium). The quantities χ and η refer to the total absorption and emission quantities (both measured in m^{-1}) respectively:

[5]For a comprehensive explanation of physical quantities and light measurements, refer to Glassner's two-volume book, *Principles of digital image synthesis* [Glassner95].

$$\chi = \kappa + \sigma \tag{2.4}$$
$$\eta = q + j. \tag{2.5}$$

The first equation states that the total absorption coefficient χ consists of a true (or thermal) absorption coefficient κ and a scattering coefficient σ representing the energy loss from out-scattering. The second equation shows that the total emission η is defined as the sum of a source (or thermal) term q representing emission and an in-scattering term j. It is important to distinguish between thermal emission and absorption, since they increase respectively, or decrease, the beam energy. Scattering causes photon interaction with a scattering center and hence energy results in moving in different directions with (in the case of inelastic scattering) different frequencies.

Note that all six quantities depend on the position in space x. This parameter is omitted to make the equations more readable. Except for the emissive scattering part j, the other three quantities depend solely on the optical material properties which are either assigned by a transfer function or given by the underlying (gaseous) model. For j, all possible contributions from incoming light directions must be considered. The basic idea is to accumulate all incident light by integrating over all directions ω':

$$j(x,\omega) = \frac{1}{4\pi} \int_{sphere} \sigma(x,\omega') p(x,\omega',\omega) L(x,\omega') d\Omega'. \tag{2.6}$$

This equation defines the accumulation of incident light $L(x,\omega')$ by integrating over all directions ω'. All contributions are not only weighted by the scattering coefficient σ, but also by the *phase function* p. This function basically represents the angle dependency of scattering and is therefore an important optical property. Assuming a normalized phase function, it is possible to rewrite Equation 2.3 including all types of interaction. It combines the previously derived terms for emission, absorption, in-scattering, and out-scattering:[6]

$$\begin{aligned} \omega \cdot \nabla_x L(x,\omega) &= -(\kappa(x,\omega) + \sigma(x,\omega)) L(x,\omega) + q(x,\omega) \\ &\quad + \frac{1}{4\pi} \int_{sphere} \sigma(x,\omega') p(x,\omega',\omega) L(x,\omega') d\Omega'. \end{aligned} \tag{2.7}$$

[6]The similarities with the rendering equation [Kajiya86, Glassner95] are obvious and of course not surprising since this equation is just a special case of the volume rendering equation restricted to surfaces.

The goal of direct volume rendering is now to determine the radiance L from Equation 2.7 for the light transfer. It is common to implement only a subset of the equation above minimizing the cost of its evaluation. For this purpose one or more terms are removed resulting in the following principle models [Engel06]):

Absorption Only. In this case the volume represents a perfectly black material that may absorb incident light.

Emission Only. The volume consists of a gas that is completely transparent but emits light.

Emission-Absorption Model. This model is commonly used and simulates a participating medium which can emit light and absorb incident light at the same time. Indirect lighting and scattering are, however, not modeled.

Single Scattering and Shadowing. Here, light coming from an external source is scattered. In the same way, shadows are produced by considering the attenuation of light incident from an external source.

Multiple Scattering. This means the complete evaluation of Equation 2.7, i.e. including emission, absorption, and scattering effects.

In practice, however, most volume renderers implement the emission-absorption model since it offers the best tradeoff between generality and efficiency. Getting rid of scattering terms, i.e. considering the thermal absorption κ and (thermal) emission q only, Equation 2.7 can be written as

$$\omega \cdot \nabla_x L(x,\omega) = -\kappa(x,\omega)L(x,\omega) + q(x,\omega), \tag{2.8}$$

which is termed the *differential volume rendering equation*, since the light transport is described by the differential change in radiance. If only a single ray is considered, parameterized by arc length s, $\omega \cdot \nabla_x I$ can be rewritten as the derivative $\frac{dL}{ds}$ leading to

$$\frac{dL(s)}{ds} = -\kappa(s)L(s) + q(s) \tag{2.9}$$

In other words, the positions and solid angles are simply substituted by the length parameter s. Using this *volume rendering equation*, the *volume rendering integral* can be derived by integrating radiance L. This is explained in the next section.

2.2.2 Volume Rendering Integral

The volume rendering integral is derived by integrating Equation 2.9 along the direction of light flow with starting point s_0 and endpoint s_n:

$$L(s_n) = L_0 e^{-\int_{s_0}^{s_n} \kappa(t)dt} + \int_{s_0}^{s_n} q(s) e^{-\int_{s}^{s_n} \kappa(t)dt} ds. \tag{2.10}$$

In this equation L_0 denotes the light entering the volume from the background at position $s = s_0$, while $L(s_n)$ denotes the radiance leaving the volume at position $s = s_n$ and finally reaching the eye point. In other words, the first term of the equation describes the light measured at the background and is hence omitted if there is no background attenuation by the volume. The second term is the main part since it represents the integral contribution of the source term attenuated by the participating medium along the remaining distances to the eye point.

Background attenuation L_0 and volume emission q need to be multiplied by a transparency factor. To compute this factor, the optical depth must be computed first. Between the positions s_1 and s_2, optical depth is defined as

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(t)dt, \tag{2.11}$$

which is basically a measure indicating how long the light travels before it is absorbed. Small values make the volume transparent while large values of τ make the volume more opaque. The corresponding transparency can be computed by plugging in the e-function:

$$T(s_1, s_2) = e^{-\tau(s_1,s_2)} = e^{-\int_{s_1}^{s_2} \kappa(t)dt}, \tag{2.12}$$

leading to a more readable version of the volume rendering integral:

$$L(s_n) = L_0 T(s_0, s_n) + \int_{s_0}^{s_n} q(s) T(s, s_n) ds. \tag{2.13}$$

In general, however, it is not possible to solve this integral analytically, and hence it is approximated by numerical methods. From the fact that points of the underlying 3D scalar field are not continuous but at discrete positions in space, it follows that the integration domain should be split into a number of smaller subsets where limits are given by the locations $s_0 < s_1 < ... < s_{n-1} < s_n$, i.e. the ith segment is determined by the interval $[s_{i-1}, s_i]$. The starting position is set to $s = s_0$ and the endpoint of the interval $s = s_n$. Note that these intervals generally do not have equal length. Using Equation 2.13 it is possible to compute the radiance at location s_i and the interval $[s_{i-1}, s_i]$ using:

$$L(s_i) = L(s_{i-1})T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s)T(s, s_i)ds. \tag{2.14}$$

By setting:

$$T_i = T(s_{i-1}, s_i), \qquad \text{and} \qquad c_i = \int_{s_{i-1}}^{s_i} q(s)T(s, s_i)ds,$$

one can write $L(s_n)$ as the product of the sum of all c_is and the product of all T_j's

$$L(s_n) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \qquad \text{with} \qquad c_0 = L(s_0). \tag{2.15}$$

Note that the transparency T_i is often replaced by its opacity, i.e. setting $\alpha_i = 1 - T_i$. What is now left to solve the (approximated) volume rendering integral is the computation of the transparency T_i and color c_i of the given intervals. One possible solution is to approximate the volume rendering integral by a Riemann sum over n equidistant segments of length $\delta x = \frac{(s_n - s_0)}{n}$. As illustrated in Figure 2.3, this is a piecewise-constant function. Each segment of the integral is approximated by its corresponding rectangle. This is defined by the function value at the sampling point and the sampling width given by the intervals. Alternatively, the segment can be approximated by assuming a linear interpolation between two discrete points (see Figure 2.3). The ith segment transparency T_i and color contribution c_i can be approximated as follows

$$T_i \approx e^{-\kappa(s_i)\Delta x} \qquad \text{and} \qquad c_i \approx q(s_i)\Delta x. \tag{2.16}$$

Unfortunately, uniform sampling may result in artifacts since a homogeneous material is assumed. As an alternative, adaptive sampling provides a better quality since non-homogeneous regions are sampled more densely compared to homogeneous regions. Another possibility is the Monte-Carlo approach, which avoids uniform sampling. Pre-integrated volume rendering [Engel01] allows for a faster evaluation, since it pre-computes all possible results from the integral and stores them in a table for lookup.

Since the volume integral along a ray is evaluated step by step, it is often referred to as *compositing*. In computer graphics, compositing means computing a weighted sum of two or more elements. The summations and multiplications in Equation 2.15 are split into several simple operations executed sequentially. The two basic approaches are *front-to-back* and *back-to-front* compositing. Setting $\alpha = 1 - T$ (i.e. opacity) and letting C represent the

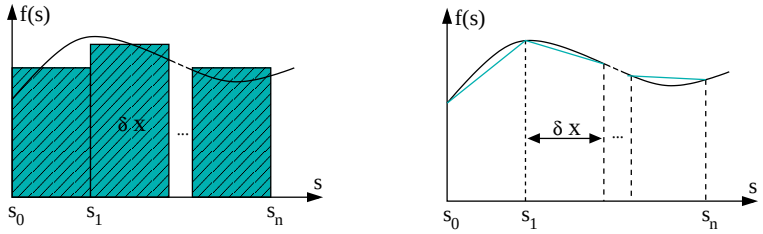


Figure 2.3: Approximating the volume rendering integral can be solved in several ways. Left: *Approximation using a Riemann sum. Instead of evaluating the integral, the area of the rectangles is summed as an approximation.* Right: *Evaluating the integral at discrete points and assuming a linear interpolation in between. Higher-order interpolation schemes are also possible. Note that Δx was chosen to be extremely large for this demonstration to visualize the difference between the methods.*

color, i.e. both the newly contributed radiance c and the accumulated radiance L are covered by this term, the *front-to-back*-operation is defined as follows:

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \tag{2.17}$$
$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \tag{2.18}$$

where subscript src is interpreted as source or input from the optical properties of the data set and subscript dst is the destination or output quantity. Hence, the output color C_{dst} is computed by adding the new input color C_{src}, weighted by the current opacity value α_{dst}. The term α_{dst} is similarly computed. Applying Equation 2.17 and 2.18 iteratively while marching through the volume updates opacity and color. Obviously, there is no contribution if the opacity reaches $\alpha_i = 1$. In this case the iterative marching can be stopped, which is commonly referred to as *early ray termination*.

Reversing the traversal direction leads directly to the *back-to-front* compositing scheme:

$$C_{dst} \leftarrow (1 - \alpha_{src})C_{dst} + C_{src} \tag{2.19}$$

Since no accumulated opacity is needed, its iterative update is not necessary when determining the color contribution. Hence the compositing is easier but does not allow for *early ray termination*. Generally, *back-to-front* rendering does not work with regard to perspective ray tracing and ray casting and will therefore not be discussed further.

Instead, this paragraph closes with briefly applying alternative approaches to this scheme. For example, maximum intensity projection (MIP) is often used in medical applications. The goal here is to find the maximum value found along a (light) ray path. This can be achieved by setting:

$$C_{dst} \leftarrow max(C_{dst}, C_{src}) \tag{2.20}$$

Obviously, the order of the iteration does not matter, allowing for random access of all values, and thus enabling the use of acceleration structures.

When rendering iso-surfaces, i.e. implicit surfaces defined by a certain value, the basic operations are a little different. Here, it is basically a boolean decision whether there exists a contribution C along the ray that is equal to the searched value. If so, an iso-surface was found, otherwise, it was not. Since the *first* intersection with the iso-surface C defined by C_{iso} is wanted, the order of traversal is important:

$$C = \begin{cases} true & C_{iso} = C_{src} \\ false & \text{otherwise} \end{cases} \tag{2.21}$$

An alternative interpretation is to look at iso-surface rendering as a special case of a transfer function, i.e. a function which results in exactly one non-zero solution.

As Engel et al. [Engel06] point out, the color C can be interpreted as radiance values for a consistent and physically correct description of light transport. If fixed camera parameters as well as a linear response to the camera is assumed, the strength of the recorded color values is proportional to the incident radiance. The final radiance C can therefore be seen as a measure for RGB values.[7]

2.2.3 Types of Volumetric Data

This paragraph discusses practical issues of volume rendering with the focus on ray tracing. In the volume rendering pipeline, data traversal is the first step, and is largely dependent on the underlying topological organization. Typical topological volume organizations are regular, curvilinear, and unstructured sets of 3D points in space. Since each organization necessarily consists of discrete points in space, the signal must be reconstructed using filter operations as described in Section 2.2.4.

Different types of volume organization result from different data sources and volume acquisition techniques. A typical field for scalar volume data is medical imaging, where the data is acquired by some kind of scanning

[7] The fact that this is not true for intermediate values is often neglected.

device. One of the most frequently used devices is computerized tomography (CT) where the physical scanning process is based on x-rays. In a tube the radiation is sent from one side to an opposite detector array while traversing the patient's body in between. Different materials, like skin and bone, result in different attenuation of the radiation recorded. The emitter and detector are then rotated around the patient's body resulting in a collection of 2D radiation 'images'. Back-projection is often used for reconstruction and works easiest with regular 3D grids.

Another device is magnetic resonance imaging (MRI) where nuclear magnetic resonance is used to identify different materials in a 3D spatial context. A rather strong magnetic field is used to align the spins of atomic nuclei. A much weaker, second magnetic field sends an excitation pulse to perturb the aligned spins. If these spins realign, radiation is emitted and recorded. Different materials in space are located using a magnetic gradient field. As for the CT and most other types of medical devices (e.g. ultrasound, positron emission tomography), the volume data set is reconstructed from the detected feedback. Both devices use scanlines and produces therefore rectilinear grids.

Semi-structured or unstructured data sets sometimes produced by Ultrasound but result more often from simulations, like computational fluid dynamics (CFD), finite element methods (FEM). The simulated data points are distributed in space to form a best fits to the applied physical simulation, i.e. adapted to the physical phenomena investigated. Besides the simulation of physics, fire and explosion simulations for special effects are also possible. As a last example, consider the case of converting the surface representation of a 3D object into a set of voxels, which is termed voxelization. It should be noted that such irregular samples often resampled in a 3D grid.

As already mentioned, each element of this field represents a scalar value. In most cases, this datatype is a 16 bit[8] integer value or a 32 bit floating point value. The following paragraphs describe three major types, their properties, as well as their decomposition into primitives.

Note that using radial basis functions do not require a topology at all, since their value only depends on the distance from the origin or some other center point. Ray tracing, however, requires in general a topology for an efficient traversal making the following discussion worthwhile.

[8]Data obtained from CT or MRI scanners usually use only the lower 12 bits, allowing the representation of values between 0 and 4095. Such measurements are expressed in Hounsfield units stating the radio density of a material. The radio-density of distilled water at standard pressure and temperature is defined as 0 (see [Hounsfield80] for more details).

2.2.3.1 Regular Data Sets

This is the most structured way of representing a volume. All scalar values within the field are arranged in a three-dimensional grid with equal spacing in each dimension, i.e. it is a uniform grid. Due to the organization in a 3D regular grid, these values are usually referred as voxels (a portmanteau of the words volumetric and pixel). Due to their inherent organization, such data is usually stored in an array. All voxel positions in space are implicitly given due to the regular (i.e. equidistant) organization of the grid.

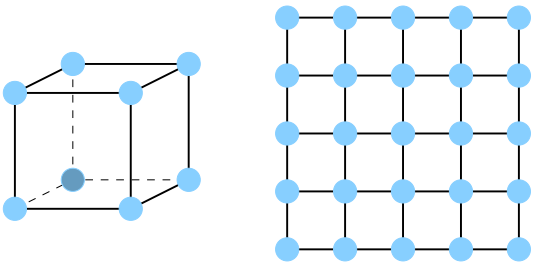


Figure 2.4: Regular Data Sets (right) contains samples organized as cubes (left). Each cube is defined by its eight corner voxels. Note that the position in space is equidistant and therefore implicitly given.

Anisotropic Regular and Rectilinear Data Sets If the spacing between the voxels is equidistant within each dimension but different for each dimension, the grid is called *anisotropic regular*. Instead of cubes, cuboids have to be handled. If this property is not given either, the grid is called *rectilinear* (see Figure 2.5), i.e. the spacing varies within the voxels in the same dimension.

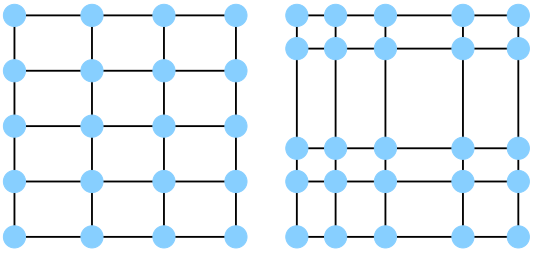


Figure 2.5: Left: *The spacing varies between the dimensions but not within each dimension.* Right: *Rectilinear data sets with different spacing.*

2.2.3.2 Curvilinear Data Sets

In so-called semi-structured grids, all values are arranged in an array, i.e. there is still adjacency information, and hence some topology implicitly encoded in the grid structure.

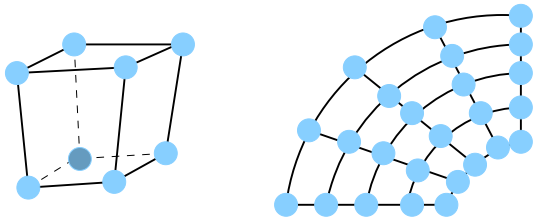


Figure 2.6: Like regular cells, curvilinear cells consist of eight neighboring values. The position of each value within a cell is arbitrary, as long as no overlapping between two cells is introduced.

The grid is, however, warped by some function which makes it necessary to store with each value a three-dimensional vector representing the position in space. Decomposing this volume into primitives still results in cells with eight corner values (see Figure 2.6). A cell, usually referred to as an hexahedral (cell), is not necessarily convex.

2.2.3.3 Unstructured Data Sets

Unstructured data sets consist of a set of points distributed in space with no adjacency information at all. It is just a list of spacial coordinates with an attached scalar value, or in short, a list of values. Since unorganized data is hard to traverse, a topological structure is often added requiring additional storage costs. This structure could be built upon an arbitrary primitive, but in most cases, a tetrahedral mesh is constructed from such a point cloud by applying, e.g. a Delaunay tetrahedralization [Choi02]. Hence the corresponding primitives are tetrahedra consisting of four values at their corners (see Figure 2.7). It is common to add further restrictions to the mesh, i.e. allowing neither holes nor overlaps of adjacent cell faces.

Tetrahedra belongs to a topological class class called *simplices* or *simplicial cells*. A *simplex*, or more precisely, a *n-simplex* is defined as the convex hull of $(n+1)$ affinely independent points in Euclidian space of a dimension equal to or greater then n. In this way, a tetrahedron is a three-simplex (see Figure 2.7), and can be used to build a triangulation of an 3D domain.

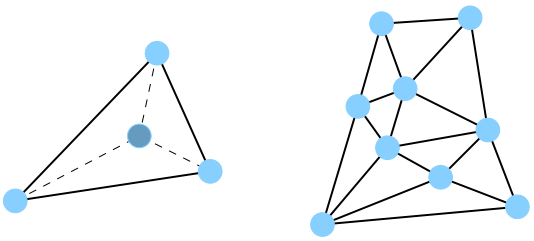


Figure 2.7: Unstructured volume data sets consist of point clouds only and therefore have no adjacency information. Since this information is needed, e.g. for interpolation, it is common to apply a Delaunay tetrahedralization [Choi02], i.e. each cell is now a tetrahedra (left) with four values. All connected tetrahedra construct a tetrahedral mesh.

2.2.4 Reconstructing a Continuous Volume Signal

Many types of volumes share the problem that there exists only a finite set of discrete values from which a continuous signal has to be reconstructed. This is a well-known problem of signal theory (refer to Oppenheim and Schafer [Oppenheim75] for a thorough discussion).

Here it will suffice to say that a linear interpolation delivers reasonably good results compared to the ideal *sinc*-function for reconstructing a continuous signal. For simplicity, a regular grid with unit lengths in all three cell dimensions is assumed. This can be easily achieved by applying a scaling factor resulting in normalized coordinates. Linear interpolation can then be computed by:

$$f(p) = (1 - x)f(a) + xf(b), \tag{2.22}$$

where $f(a)$ and $f(b)$ are sample points at the spatial coordinates a and b respectively. This scheme can be easily extended to higher dimensions by applying a tensor product-like reconstruction, which is inherently supported when using regular grids. As depicted in Figure 2.8, this is achieved by separating the interpolation across the dimensions, i.e. by applying a sequence of linear interpolations. This can be extended to the 2D case by first interpolating between a and b, then between c and d, followed by a third interpolation between both interpolated values

$$f(p) = (1 - y)f(p_{ab}) + yf(p_{cd}), \tag{2.23}$$

assuming that the sampled points in the y-dimension are given by c and d. In the same way, it is possible to add the last dimension for the 3D case

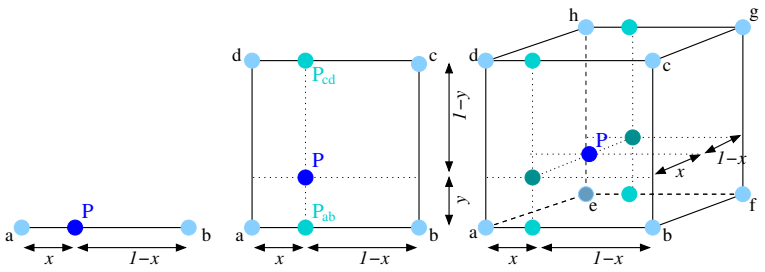


Figure 2.8: From left to right: *Linear interpolation between points a and b. Interpolating between points c and d and applying one subsequent interpolation results in a bilinear interpolation (middle image). Based on similar ideas, a tri-linear interpolation can be derived from the points a - h (right image).*

$$f(p) = (1 - z)f(p_{abcd}) + zf(p_{efgh}), \tag{2.24}$$

which results in a total of seven linear interpolations. Using substitution, this leads to a rather large equation [Shirley05]. It is also clear that this trilinear interpolation involves terms up to cubic order which makes tri-linear interpolation non-linear with respect of the polynomial to be solved. In Section 4.3.3 it will be shown that this is an important fact for recovering an implicit function (see Section 2.2.5).

Due to the irregular structure of the more general hexahedral, it is tedious to derive a cubic polynomial here, but still possible [Pascucci00, McDonnell04]. An alternative approach will be discussed in Section 6.4.4 using a combination of bilinear patch intersections and a linear interpolation.

On the other hand, with a three-simplex (i.e. tetrahedron) this interpolation is rather easy to compute. The well-known 2D barycentric coordinates (see [Shirley05]) can be extended to 3D barycentric coordinates. Instead of putting three triangular sub-areas into a relation, four tetrahedral subvolumes are related to each other:

$$\begin{aligned} f(p) = (1 - \beta - \gamma - \delta) * f(V_{qbcd}) + \beta * f(V_{qacd}) + \\ \gamma * f(V_{qabd}) + \delta * f(V_{qabc}), \end{aligned} \tag{2.25}$$

if V_{qbcd}, V_{qacd}, V_{qabd}, and V_{qabc} are four (partial) volumes of the tetrahedron defined by the vertices a, b, c, and d and a point $q = (x, y, z)$. The hit point is within the tetrahedron if and only if $\beta > 0$, $\gamma > 0$, $\delta > 0$, and $\beta+\gamma+\delta < 1$.

2.2.5 Volume Rendering Techniques

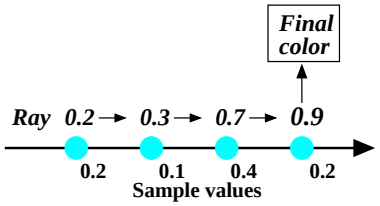
It is now time to discuss semi-transparent rendering, maximum intensity projection, and iso-surface rendering from the perspective of an actual ray casting implementation. For the theoretical implications, refer to Section 2.2.2.

2.2.5.1 Semi-Transparent Rendering

This is a direct implementation of the volume rendering integral taking emission and absorption into account but ignores scattering. Neglecting scattering provides a good tradeoff between generality and efficiency of computation. The contribution along a ray is computed by solving the volume rendering integral (see Equation 2.7), which is implemented iteratively either *back-to-front* or *front-to-back*. For ray tracing, the usual way is a *front-to-back* implementation since rays traverse the volume in this order. The corresponding pseudo-code and illustration for an approximation is stated below in Algorithm 2.

Algorithm 2 Semi-Transparent Rendering.

for $i = 1..N$ **do**
  $color := color + (1 - alpha) * \rho_i$
  $alpha := alpha + (1 - alpha) * \alpha_i$
end for
return *color*



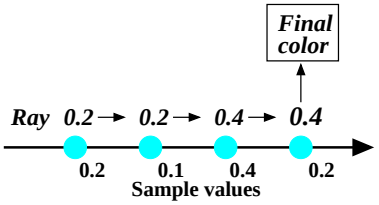
In this pseudo-code, *alpha* and *color* are accumulated values of opacity and color. In contrast to *back-to-front* rendering, the opacity must be tracked independently. Intensity ρ_i (simply speaking the intermediate color value) is directly obtained from the volume, i.e. by reconstructing a continuous signal (see Section 2.2.4) using interpolation and mapping. The opacity value is set to $\alpha := 1 - e^{-D}$ (i.e. $(1 - T)$) or simply $\alpha = -D$ (neglecting the e-function for small D), with D as distance between two adjacent samples along the ray. It is also possible to apply early ray termination, i.e. the loop stops after $alpha = 1.0$ respectively $alpha = 1.0 - \epsilon$ is reached (not shown in pseudo-code).

2.2.5.2 Maximum-Intensity-Projection (MIP)

Here, the volume is traversed as previously described. Instead of accumulating values, however, only the maximum value is of interest. This method is often used for Magnetic Resonance Angiograms where thin structures, e.g. blood vessels etc. must be rendered accurately.

Algorithm 3 Maximum-Intensity Rendering.

for $i = 1..N$ **do**
  $color = \text{MAX}(color, \rho_i)$
end for
return *color*



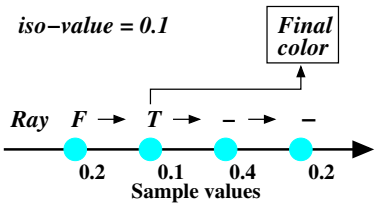
No opacity or any other value needs to be tracked (see Algorithm 3). It is possible to accelerate the search for the maximum value by using appropriate structures. Parker et al. [Parker99b] store the maximum values in a multi-level grid and creates a priority queue for each ray traversing the volumetric grid. Regions with a lower maximum value compared to the highest value in the priority queue can be efficiently skipped. A similar acceleration structure can be used for the next method, called iso-surface rendering.

2.2.5.3 Iso-Surface Rendering

This techniques aims at rendering a surface which is defined by some function $f(x, y, z) = \rho$. While ρ is a (user-defined) iso-value, $f(x, y, z)$ is usually derived from the reconstruction. For example, using a tensor-product like interpolation for regular grid (see Section 2.2.4), it would be a cubic polynomial. A particularly important application for this rendering method is virtual endoscopy.

Algorithm 4 Iso-surface Rendering.

for $i = 1..N$ **do**
  if $\rho_i = \rho_{iso}$ **then**
    return *true*
  end if
end for
return *false*



Algorithm 4 assumes that the value can be find among the number of samples N. The user-defined value ρ_{iso} is compared with each value ρ_i along the ray. In practice, it is necessary to reconstruct a continuous signal, e.g. using an interpolation scheme. It suffice to check the ray at intervals defined by the underlying primitives (cubes, hexahedra, or tetrahedra). Only those intervals have to be investigated further, where ρ_{iso} lies between the cell's minimum and maximum values (see Section 4.3.3).

2.2.5.4 Decomposition

Such methods visualize certain subsets of the scalar field, i.e. slices, particular points, or small geometric objects [Theisel01]. For example, a slice could be represented as a height field by interpreting the scalar values as a vector perpendicular to the slice [Nielson90]. Since the usage of such methods is limited, it will not be considered further, but is mentioned for the sake of completeness.

2.2.6 Volume Rendering Pipeline

Although the suggested approaches for volume rendering differs with respect to hardware usage and traversal order (object or image) the underlying rendering process is often implemented as a pipeline. It is common to distinguish between six stages (see [Engel06]): data traversal, interpolation, gradient computation, classification, shading and illumination, and compositing. The following provides a brief description of all stages, as well as references to upcoming chapters where specific stages will be further discussed.

Data Traversal. This is an essential first step for volume rendering. Traversing the volume data depends strongly on the underlying organization of the 3D scalar field (see Section 2.2.3). The samples found along a light ray are the basis for discretization of the continuous volume rendering integral. Chapters 4, 5 and 6 will mainly discuss how these different topologies can be efficiently traversed. However, the light rays do not hit the points within the discretized field directly, which leads to next step.

Interpolation. Since the sampling positions usually differs from the points in the 3D scalar field, the values on the light ray need to be interpolated using given scalar values close to the actual sampling position. More on this can be found when discussing reconstruction methods in general (Section 2.2.4) and reconstructing the iso-surface within a regular grid (see Section 4.3.3) or irregular grid (see Section 6.3.3 and 6.4.3).

Gradient Computation. Computing the gradient of the scalar field at sample positions increases the rendering quality by adding visual cues. This allows the use of directional light or Phong shading [Phong75] but also advanced rendering effects like global illumination. Typically, the gradient is approximated by computing central differences (see Section 4.3.4).

Classification. The properties of the traversed data set is mapped to the optical properties of the volume rendering integral. The interpolated scalar values are usually mapped using a transfer function, i.e. different materials with certain ranges of scalar values are assigned parameters in the volume rendering equation, e.g. different colors. See Section 4.5 for more information about transfer functions.

Shading and Illumination. It is possible to add an illumination term to the emissive term of the volume rendering integral, i.e. light from an external light source is considered for single-scattering effects. More common is the shading computation of an implicit surface, the so-called iso-surface (see Section 2.2.5.3). This is a special transfer function which evaluates for exactly one (discrete) input value to one and zero for all others. Once this intersection point is found and the gradient is computed, all kinds of shading effects can be applied, e.g. Phong shading [Phong75] with several light sources or even global illumination.

Compositing. This step is necessary whenever multiple contributions need to be combined, i.e. it is only required for semi-transparent rendering. The previous section already covered two basic approaches depending on the traversal order: *front-to-back* and *back-to-front* compositing.

Some parts of this pipeline work as local operators and appear therefore not necessarily in this order: interpolation, gradient computation, shading, and classification. These stages will be covered only briefly, while more attention will be given to data traversal.

2.3 Conclusion

This chapter discussed essential concepts of ray tracing and volume graphics. Ray tracing is an image-order algorithm that selects for each screen pixel the nearest object as perceived from the viewer's eye point. It was shown that a basic implementation is rather easy and the concept is powerful enough to be extended with advanced shading and lighting effects.

The subsequent section first described the transport of light and how a participating medium interacts with a light beam leading to the volume rendering equation from which the volume rendering integral can be derived. The most commonly used model is based on emission and absorption only.

Depending on the origin of the data (devices, physical simulations, etc.), structured, semi-structured, and unstructured data have to be distinguished, as the next section showed. No matter how a 3D scalar field is organized,

Property	Regular / Rectilinear (structured)	Curvilinear (semi-structured)	Tetrahedralized (unstructured)
Adjacency	implicit	implicit	explicit
Primitive	Cube / Cuboid	Hexahedron	Tetrahedron
Interpolation	tri-linear	tri-linear	barycentric / linear
Storage	scalars	scalars + positions	scalars + positions + topology

Table 2.1: This table summarizes some properties of the most often used volume data organization with respect to adjacency, primitive, interpolation and storage.

it always consists of discrete values in space, from which a continuous signal must be reconstructed to solve the volume rendering integral. Table 2.1 summarizes the most important properties for each of the discussed types, including the applicable interpolation scheme.

As demonstrated with pseudo-code examples in the following section, ray tracing is well-suited for the three most common volume visualization techniques. Figure 2.9 shows sample renderings of the engine data set (regular grid) using semi-transparent rendering, maximum intensity projection, and iso-surface rendering. The last section gave a brief overview of the volume rendering pipeline in which traversal was identified as a first and crucial step for volume graphics.

The next chapter will discuss a variety of alternative rendering approaches, such as cell projection, slice projection, splatting (vertex projection), and shear-warp. All of these methods work in so-called object-order. Such algorithms iterate over all primitives in the scene and accumulate each contribution. This process can be seen as the inverted approach to image-order algorithms. The following chapter will give an overview over major approaches for object-order algorithms and explain their shortcomings compared to the image-order ray-tracing algorithm.

Before going into details here, it is worth mentioning that a third class of algorithms restricted to iso-surface rendering exists. A popular example is the Marching Cube algorithm proposed by Lorensen and Cline [Lorensen87]. Considering binary iso-values only leads to 256 possible iso-surface alignments for the eight corner values of a cell. However, most possibilities differ only in cell rotation leading to fifteen generic types with respect to the iso-value distribution. Based upon this distribution, polygonal surfaces are placed into

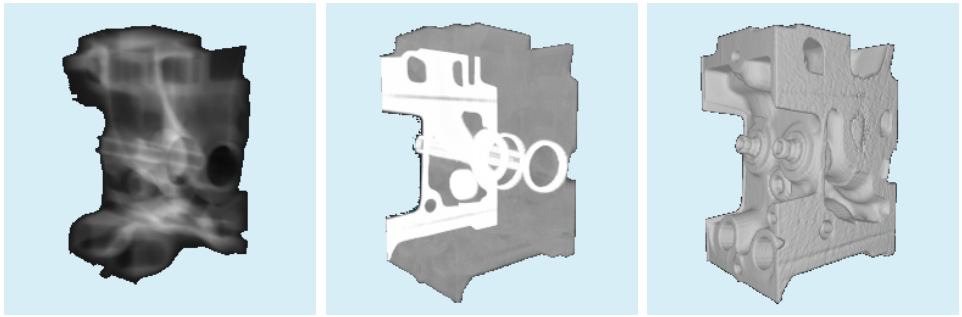


Figure 2.9: The engine is a small data set based on regular grids with an 8 Bit scalar value. From Left to Right: *Semi-transparent rendering of the density distribution, maximum intensity projection, and rendering of the iso-surface at* $\rho_{iso} = 118$.

each cell before 'marching' to the next one. In other words, the iso-surface for a specific value is extracted and can be rendered by using a rasterization algorithm implemented in graphics hardware. The main disadvantage is that this extraction step is costly and must be repeated whenever the iso-value changes. Though the algorithm was extended in many ways [Bhaniramka73, Banks03, Nielson03, Nielson04] and also adapted to tetrahedral meshes [Cignoni96]. Modern GPUs, on the other hand, allow porting more and more parts of the algorithm to the GPU [Pascucci04, Klein04], but some problems still remain. For example, the number of triangles can tremendously increase to more than one billion for large data sets, which is challenging even for today's GPUs.

Chapter 3

Alternative (object-order) Approaches

"I understand HOW. I do not understand WHY.

"1984"

This chapter will briefly cover the most relevant object-order approaches in the area of volume rendering. Projection is one of the most often implemented object-order algorithms and can be further categorized into cell-projection, vertex-projection (splatting), and texture-mapping. Software as well as hardware implementations exist for this approach. Since graphics hardware is becoming more and more flexible, recent implementations even rely completely on the GPU. Sometimes, even a combination of object-order and image-order algorithms can be found. The chapter will close with the shear-warp factorization and custom hardware implementations.

3.1 Cell Projection

In cell projection, the volumetric primitives, either cubes/cubiods, hexahedra or tetrahedra, are projected onto the image plane (see Figure 3.1). Except for maximum intensity projection (see Section 2.2.5.2), this implies an additional sorting step providing the correct visibility order of primitives as observed from the eye point.

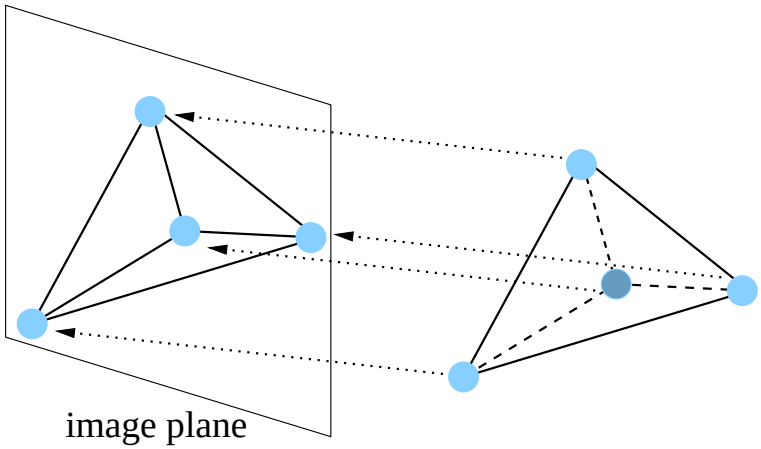


Figure 3.1: Using projection, all primitives are projected onto the image plane before processing. In this example, a tetrahedron is projected onto the image plane resulting in the loss of depth information if the z-value is not maintained otherwise.

Despite the large restrictions in the first years of dedicated graphics hardware, one of the first algorithms for projecting cells of unstructured grids

was proposed as early as 1990 by Shirley and Tuchman [Shirley90]. In this approach, all tetrahedra are sorted *front-to-back* first with respect to the eye-point. After projecting each tetrahedron onto the image plane, its outline is decomposed into one to four triangles determined by the crossings of the edges of the projected tetrahedron. For example, the tetrahedron shown in Figure 3.1 would be decomposed into three triangles. Ray integration occurs by computing opacity and color (i.e. intensity contribution) at the thickest point for each tetrahedron, followed by a linear interpolation between the triangle faces pierced by the ray. A final *back-to-front* accumulation using the *painter's algorithm* [de Berg00] provides the rendered image. Rendering times were several seconds, even for small data sets and viewport sizes.

Röttger et al. [Röttger00] additionally store color and opacity in a 3D texture map computed during pre-processing. This enables the concurrent use of transfer functions since they can be directly encoded in the texture map. By using *back-to-front* compositing, each vertex of each projected tetrahedron can be assigned all associated triangles and can blend them into the frame-buffer. However, the use of 3D textures is memory-consuming; therefore the approach has also been adapted for 2D texture maps. To this end, all dependencies of the volume rendering integral with respect to the length between two sample points are approximated in linear terms and stored in the first 2D texture map. The remaining parts of the integral depend only on the sample values and can thus be organized in another 2D texture map. Interactive rendering times were achieved on an SGI Octane MXE with up to 5 fps. This algorithm was extended in many ways, e.g. Guthe et al. [Guthe02] added hardware-accelerated pre-integration [Engel01].

3.1.1 Parallel Cell Projection

An early and straightforward parallel implementation of a projection-based algorithm was suggested by Lucas [Lucas92]. He described a pure software system that renders volumes in two passes. The system can handle all major types of topologies (see Section 2.2.3). The first pass takes care of per-vertex calculations, e.g. point and normal transformation, lighting, and avoidance of multiple processing of shared vertices. This part is parallelized by partitioning the object space.

During the second pass, which is parallelized in screen space, all primitives are scan-converted using the transformed and lit vertices from the first pass. This scan-conversion is implemented as a z-buffer algorithm (i.e. reverse of the *painter's algorithm*, see [de Berg00]). For volume rendering, each primitive is decomposed into its faces, which are then sorted *back-to-front* (in contrast to Shirley and Tuchman [Shirley90], which sort the primitives themselves).

Each face of this sorted list contains the linearly interpolated vertex color, opacity, and depth value for each pixel in the projection. Lucas used only the centroids of each face as a sorting criterion which may sometimes lead to the incorrect sorting. Lucas reported no rendering times for his volume renderer but showed at least that his surface renderer can be implemented with linear speed-up with respect to the number of processors.

Wilhelms et al. [Wilhelms96] use scan conversion too for their software renderer. The volumetric primitives are decomposed into triangles depending on the grid type, e.g. twelve triangles for a hexahedra. The scan-line conversion uses y- and x-bucket lists which consist of the triangles first appearing on the scan-line and allow the use of coherence between adjacent pixels as well as scan-lines. These lists are accumulated *front-to-back* for color and opacity. Vertex transforming and bucket sorting can both be parallelized. Additionally a simple hierarchical scheme based on kd-trees was introduced, which merges a certain number of polygons based upon a user-defined threshold. This approximation is used for distant views, while the original data is traversed for close views. As a second advantage, the kd-tree also culls invisible regions from the current viewpoint. Rendering images with a 512^2 viewport nevertheless takes up to one minute and more on the SGI Onyx with four processors.

Later, Williams et al. [Williams98] developed a rather sophisticated software volume renderer based on cell projection. Although graphics hardware was used for acceleration, their system was not intended for interactive purposes but to produce high-quality images in batch mode. Their *HIAC* system is able to handle any unstructured data set whose cells are tetrahedra, bricks, prisms, pyramids, or a combination of these primitives. Meshes may be non-convex or even disconnected. The goal was to present a benchmark system for comparison for which all mathematical operations are described in detail. Bennet et al. [Bennett01] later parallelized the HIAC system by distributing the sorting step of the visibility ordering among several nodes. Kd-partitioning ensures load balancing of the volumetric data set. Still, this improved system needs over two seconds even when employing all 128 processors of the SGI Onyx2.

Ma and Crockett [Ma97] also distribute the data for load balancing and use MPI for distributed rendering. In contrast to the previously discussed approaches, adjacent tetrahedra are not clustered together in the same partition, but rather scattered over all available nodes. The idea is to allow good load balancing even for close views. A global kd-tree restores the ordering during rendering. Each node clusters all cells traversed by the assigned rays before sending them back to the render nodes. Scan-lines are interleaved among the rendering nodes to improve load balancing.

3.2 Hybrid Algorithms

It is also possible to project irregular meshes onto screen space first, but then process them in image-order using conventional ray casting. Bunyk et al. [Bunyk97] proposed such a method for irregular grids. In a preprocessing step each volumetric primitive, i.e. a tetrahedron (see Section 2.2.3), is decomposed into its faces (triangles). All triangles are subsequently projected onto screen, allowing for the determination of the visibility order in screen space. Casting a ray through those triangles results in a depth-sorted list. Since each triangle is associated with a tetrahedron, all traversed tetrahedra along a ray can be fetched by stepping through all triangles. This is necessary for obtaining the scalar values. Assuming a linear opacity and intensity contribution for each tetrahedron makes volume rendering possible by just linearly interpolating between adjacent faces. The main advantage of this approach is that the intersection test is reduced to a 2D problem which can be efficiently solved. However, the list of triangles per pixel needs to be updated whenever the viewpoint changes. The SGI Power Challenge machine needed eleven seconds for a 256^2 and up to 100 seconds for a 1024^2 viewport rendering. The sorting step on each viewpoint change is around two seconds, which is negligible even for larger resolutions (greater than 512^2).

Hong and Kaufman [Hong98] also suggested a projection-based ray-casting algorithm for curvilinear grids. The first face along the ray observed from the viewpoint is found by scan converting all boundary faces onto the image plane before depth-sorting along the given viewing ray. All faces of a hexahedron are decomposed on the fly into twelve triangles. To traverse the cells, all twelve faces are projected onto the image plane for intersection calculation. The interpolated scalar values along a ray can be computed together with the depth as a part of the ray-triangle intersection test. In the original approach, the remaining eleven triangles (one is already known when a ray enters the hexahedron) were tested sequentially until another intersected triangle, and hence the face where the ray exits the hexahedron, was found. In a follow-up paper, Hong and Kaufman [Hong99] suggested to group these remaining eleven triangles and project them onto the image plane. The exiting triangle can then be found by applying a ray-crossing technique [Haines94]. Accordingly, a point P is inside a 2D polygon if and only if the horizontal ray starting from P and shooting infinitely to the right crosses the polygon edges an odd number of times. By checking how many times this horizontal ray crosses the edges of tested triangles, the exiting triangle, and hence the exiting face, can be determined. This new algorithm is twice as fast as the original implementation, but still needs 3.45 seconds for a 300^2 viewport rendering of the *Blunt-fin* data set (SGI Octane, 198 MHz MIPS processors).

Another hybrid approach was suggested by Weiler et al. [Weiler03]. They implemented an efficient ray-caster for tetrahedral meshes on a consumer graphics card. The first tetrahedron along the ray is found by rasterizing the extracted boundary faces of a given model. For their ray-casting approach, a ray-plane intersection is used [Garrity90] to determine the exiting face of the currently processed tetrahedron. The ray integration relies on pre-integration, as described in [Engel01]. All of these computations are performed in the fragment program. Due to the limited flexibility of graphic boards, it is impossible to trace a ray completely trough the volume. Instead, multiple rendering passes are necessary, one for each tetrahedron along the ray. The pre-computation leads to a total memory requirement of 160 bytes per tetrahedron, and hence the size of the model is restricted to 600,000 tetrahedra on the card used (ATI Radeon 9700, 128 MB RAM). Additionally, a convexification needs to be applied as suggested by Williams et al. [Williams92], which further increases the number of tetrahedra of the data set. Interactive rendering of mid-sized models is possible with 2 to 5 fps. A more compact data set representation taking advantage of implicit neighbors [Weiler04] achieves similar performance with less memory consumption.

3.3 Object-Order Ray Casting Algorithms

Object-order ray casting restricted to regular grids was also developed by several researchers. Mora et al. [Mora02] suggested a software implementation enabling interactive frame rates for mid-sized data sets (e.g. bonsai, engine, etc.). Since they use an orthogonal projection, they can take advantage of the fact, that every cell projection corresponds to the same hexagon template except for the translation. Pre-computed min/max octrees are used for skipping empty (transparent) regions within the volume. Sub-volumes of m^3 voxels are then projected onto the image plane. The implemented iso-surface renderer uses hierarchical occlusion maps [Zhang97], i.e. images of different sizes indicating whether a ray have to be traversed or the previously computed value can be re-used respectively.

Hong et al. [Hong05] adapted a simular approach to current graphics hardware. Fragments are generated corresponding to the rays intersecting that cell. The correct order between sub-volumes is implicitly given by the min/max octree. Dividing each sub-volume into pre-computed layers further reduces the visibility ordering.

Another iso-surface rendering algorithm also based on object-order ray casting was proposed by Neubauer et al. [Neubauer02]. The entire data set is subdivided into *macro-cells* of size m^3 where m is usually between four

and ten. These macro-cells are then used to build a min/max octree similar to [Wilhelms92]. For each pixel on the image plane that has not yet been processed, the octree is traversed and at each traversal step, the min/max values are checked for whether boundary cells (i.e. cells possibly containing the iso-surface) are in the sub-tree or not. At a leaf node, the boundaries of the macro-cell are projected and rasterized onto the image plane yielding a hexagonal footprint. For each pixel in this hexagon, *local rays* are then used to traverse the macro-cell grid. This reduces the number of traversal steps for the octree structure since all pixels that are covered by the hexagon would perform the same traversal steps. For the macro-cell traversal, the method of Amanatides and Woo [Amanatides87] is used. If a boundary cell is encountered, an intersection test is performed with the iso-surface and if true, normal and shading calculations are performed.

3.4 Vertex Projection (Splatting)

Splatting is a forward mapping algorithm, i.e. the contribution of each voxel to the final image is calculated independently. The contribution of a voxel in object space that is projected on the image plane is called a *footprint*. Hence, this footprint is in fact the reconstruction kernel that represents the original signal determined by the voxel in object space. All footprints along a ray are accumulated to obtain the final pixel color either in *front-to-back* or *back-to-front* order. Splatting was first described by Westover [Westover90], which concerned rendering regular grids on a CPU. He proved that this footprint does not depend on the actual position of the voxel in space, enabling the use of look-up tables for an approximation.

Besides the previously described compositing of all splats, there also exists the so-called sheet-buffer method [Müller98]. Here, the splats are organized in cache-sheets that are aligned parallel to the volume face most closely parallel to the image plane. Each sheet-buffer is first composited into a cache image by traversing the volume *back-to-front*, i.e. the voxel contributions are added slice-by-slice. To avoid popping artifacts, which occur when the orientation of the sheets suddenly change, the sheet-buffer is always arranged parallel to the image plane [Müller98]. Since this new buffer does not correspond to the voxel positions in space, the new positions must be computed using interpolation so that they lie on the slice again. Whenever a sheet buffer has received all contributions, it is composited with the current image and the next slice is processed.

This approach works best for orthographic views since it requires in this case only a single footprint table and reconstruction kernel, which is constant

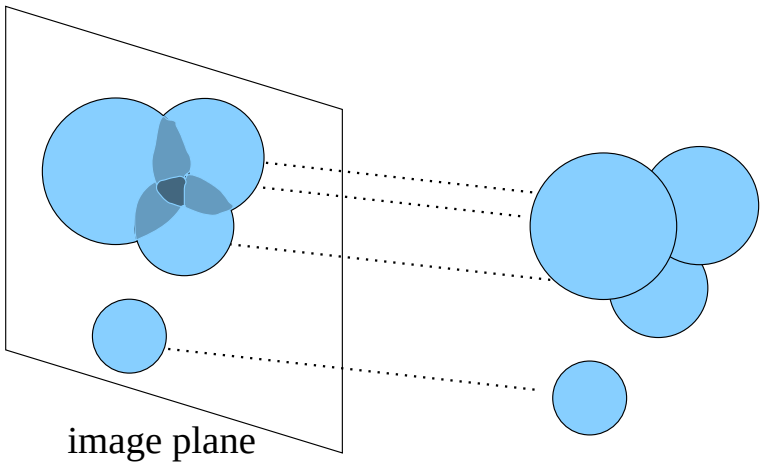


Figure 3.2: Vertex projection interprets the scalar values (illustrated here as varying point sizes) as a spatial extent (i.e. footprint) projected onto the image plane. Accumulating these footprints per pixel yields the final color.

except for the screen space offset for every voxel. Müller and Yagel [Müller96] therefore suggested a hybrid method. The voxel contributions are partly precomputed by splatting in object space. However, pixel accumulation occur by shooting rays intersecting the splats in space, similar to ray casting. This enables at the same time other optimization techniques used for ray tracing, e.g. space leaping, adaptive screen sampling or spatial acceleration structures. Performance was stated with 30 seconds for the *MRbrain* data set rendered with a 260^2 viewport on an SGI Indigo with 200 MHz.

Zwicker et al. [Zwicker01] addressed the problem of aliasing effects caused by incorrect visibility determination during *back-to-front* compositing, since the reconstruction kernels are assumed to be non-overlapping. Note, that these kernels must overlap to avoid rendering artifacts. Such aliasing effects can be reduced by adapting Heckberts elliptical weighted average (EWA) resampling filter [Heckbert89] for volume splatting. The footprint function is replaced with a re-sampling filter. Each footprint function is now separately band-limited and hence respecting the Nyquist frequency of the rasterized image. They chose an elliptical Gaussian kernel as basis functions and a low-pass filter for anti-aliasing. These quality enhancements lead to a rendering time of eleven seconds for the *Skull* data set on a system equipped with a 866 MHz Pentium-III.

Unfortunately this makes the computation of the footprint rather expensive, leading to the idea to simplify this computation in an adaptive

way [Chen04]. The key observation is that rays diverge if the volume data is far from the viewpoint, thus making the sampling rate of these rays fall below the sampling rate of the volume data set. In this case, the low-pass filter is the dominant component, while for close range volume data, the reconstruction filter dominates. By classifying and processing each volume particle with respect to the previous scheme, the computation of the footprint is accelerated without reducing the quality of EWA splatting. Additional performance is gained from a GPU implementation allowing for interactive frame-rates of mid-sized data sets.

Performance was also the main focus of recent implementations. For example, Jang et al. [Jang04] added an octree to locate the rational basis functions (RBFs) intersecting the cache-sheets. For each sheet plane pixel, a fragment program evaluates the exponential Gaussian function of the RBFs. Frame rates vary largely, for example 7 to 70 fps for the *Blunt-fin* rendering with a 400^2 resolution, which is caused by the large variation of the RBFs. The testing system was equipped with a 2.8 GHz Pentium-IV and an nVidia GeForce FX 5900. Recently, Neophuytou et al. [Neophytou06] used the floating-point rasterization facilities of the latest graphics hardware to avoid invoking expensive fragment programs. Both approaches works on irregular data. Near-interactive frame rates (1.6 fps for the *Blunt-fin*) were achieved on a Pentium-IV with 3.6 GHz and an nVidia QuadroFX 3400.

3.5 Texture Mapping

Cabral et al. [Cabral94] was one of the first to show that texture capabilities of graphic boards can be used directly for rendering volumetric data sets. It can be seen as a hybrid approach of backward and forward projection. In a first step, slices are generated parallel to the image plane by trilinearly interpolating the sample values on each slice (i.e. backward projection). After a slice has been processed, the result is blended into the frame buffer (i.e. forward projection). A final attenuation handles the case of off-center pixels, where the path length differs.

Engel et al. [Engel01] improved rendering quality using *pre-integrated* volume rendering. The ideas presented in [Röttger00] are extended and improved upon for regular grids. Pre-integrated classification overcomes the problem of high Nyquist frequencies resulting from non-linear transfer functions. Instead of applying higher-order interpolations or adaptive sampling, the idea is to split the numerical integration into two parts. One handles the continuous scalar field and the other handles the transfer function. In the first step, the scalar field is sampled along the viewing ray. This sampling

has its own Nyquist frequency that is independent of the transfer function. Since this integration is approximated by a Riemann sum (see Section 2.2.2), the sampled values define a one-dimensional piecewise linear scalar field.

Storing these values in a table reduces the integration step to a table lookup, with the (interpolated) scalar values, as well as the length at the start and end of the associated ray segment. Since the transfer function is directly encoded in this lookup table, the table needs to be refreshed whenever this function is modified by the user. Assuming constant ray segment lengths and local updates of the transfer function improves the performance. Using a 200 MHz CPU together with an nVidia GeForce3 produces a performance of 4 fps for a 256^2 viewport. Röttger et al. [Röttger03] later combined this approach with volumetric clipping and advanced lighting effects.

The approaches discussed so far compute all scalar values of the grid for rendering the volume, regardless of their visibility. Li et al. [Li03] proposed therefore to partition the volume into smaller sub-volumes with similar properties. These properties depend on the transfer function, i.e. scalar values within a certain range are grouped together. A kd-tree is used to render this partitioned volume with correct visibility order, where each node in the tree is associated with a sub-volume. Each sub-volume is culled and clipped against an opacity map. This opacity map corresponds to a region of the frame buffer and stores the minimum opacity of the frame buffer pixels found within that region. They reported 10 fps for the *Engine* data set using a 2.5 GHz Pentium-IV and a nVidia GeForce4 graphics adapter.

3.6 Shear-Warp

Shear-warp [Lacroute94] is still one of the fastest software implementations for volume rendering. In contrast to the algorithm presented by Drebin et al. [Drebin88], the number of resampling passes is reduced from three to two. The basic idea is to factorize the projection matrix into a 3D shear and 2D warp. Shearing transforms the data set into sheared object space. In this space, all viewing rays are parallel to one of the orthogonal axis. The volume is considered as a stack of 2D slices. The 2D slices are then aligned and re-sampled such that they are all perpendicular to the viewing direction which simplifies the traversal of the volume significantly. Finally, this intermediate image is then warped to the image plane (see Figure 3.3) to correct the shearing. Perspective Rendering requires individual scaling of each slice during re-sampling. This original implementation needs one second for orthographic projection and three seconds for perspective projection on an SGI Indigo R4000 rendering a 256^2 viewport.

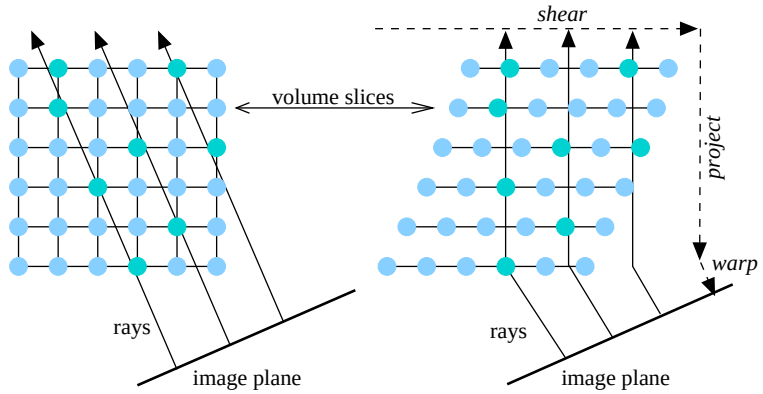


Figure 3.3: Instead of shooting rays from the view plane, the volume slices are sheared so that all rays are perpendicular to the slices, which simplifies the traversal tremendously. This sheared image must be warped in a second step allowing for a correct display of the rendered volume. For Perspective rendering, each slice needs an individual scaling factor for re-sampling.

Rendered images are prone to show stair-casing artifacts near a 45° viewing angle. Intermediate slices lying halfway between two adjacent volume slices partly avoid this problem. Furthermore, images may blur during a zoom-in since the re-sampling of the warp matrix is not adaptive. An enhanced version, solving these problems can be found in [Sweeney02], but at an increased computational cost. Although the warping step significantly limits the image quality, especially for perspective rendering, it is still used today, e.g. for the *VolumePro* [Pfister99] described in the upcoming section.

3.7 Custom Hardware

The need for custom graphics hardware arose with the demand for real-time volume rendering systems. Neither GPUs nor CPUs were fast enough at that time to achieve this goal. Most systems have been developed for rendering regular data, e.g. *Cube* [Kreeger99], *Vizard* [Knittel97], and *VolumePro* [Pfister99]. Due to the highly regular computation, all of them achieved real-time or at least interactive frame rates.

Cube [Kreeger99] implements a hybrid algorithm using Shear-Warp; however, it was designed to compute only the shear-step on-board and let the graphics board warp and render the image. Eight identical rendering pipelines are able to render a 256^3 volume at 30 fps. Never commercially realized, *Cube* [Kreeger99] was the predecessor of the *VolumePro* board [Pfister99].

Although the scalability was enhanced, perspective rendering was still not possible. The latest generation consists of separate sample and voxel processing pipelines. Voxel processors traverse data slice-by-slice in memory order and store them in on-chip buffers. These buffers are traversed by sample processors responsible for illumination, filtering, and compositing. More interestingly, perspective rendering is now possible [Wu03].

Vizard [Knittel97] and *Vizard II* [Meissner00] were both based on an image-order algorithm offering full ray casting including early ray termination. Phong shading [Phong75] was implemented using lookup tables. The performance is not comparable to *VolumePro* due to the FPGA implementation. The system is therefore more flexible but lacks in rendering performance (10 fps for 256^2 viewport).

Changing or extending custom hardware, however, is tedious and costly. Another disadvantage which custom hardware shares with GPUs is limited memory. Out-of-core solutions are not, in general, an alternative due to the high bandwidth needed.

3.8 Conclusion

The methods discussed in this chapter all provide fast and reliable volume rendering. Parallelization is possible in almost all approaches, which works favorably in conjunction with modern GPUs. As it will be shown in Section 4.1, however, the flexibility of modern graphic boards allow the implementation of ray casting directly. This significantly improves the image quality while preserving speed. Previous generations of graphic boards supported only cell projection directly, which requires a visibility sorting of primitives whenever the viewpoint changes. This is also true for hybrid approaches. Splatting is fast and memory efficient, but slow for perspective rendering and prone to rendering artifacts. Using texture mapping, the data set size is restricted to the limited on-board texture memory. Shear-Warp is also not well-suited for perspective projection, prone to rendering artifacts, and not very memory efficient since each volume needs to be stored three times (one per dimension). Custom hardware is fast and delivers high quality images, but offers limited flexibility.

Scientific visualization, however, demands both high quality *and* flexibility. Ray tracing offers both but is still considered too slow for interactive purposes[1]. The following chapters will show that this is no longer true. In fact, there exist a large variety of approaches especially for regular grids.

[1]The obvious idea of rendering with ray tracing only those parts of the image inspected by the user fails due to the unpredictable behavior of humans eye movements [Marmitt02].

This is true even for modern graphic boards allowing for a ray casting implementation. After discussing previous work at the beginning of each chapter, the main part of each chapter will focus on two specific implementations. Regular grids, *implicit kd-trees* and their usage for iso-surface rendering, as well as some extensions, will be discussed throughout Chapters 4 and 5, while in Chapter 6 a new incremental traverser based on *Plücker coordinates* and suitable for both unstructured and semi-structured grids will be explained.

Chapter 4
Static Regular Data Sets

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!', but 'That's funny ...'.

Isaac Asimov

Regular data sets consist of equidistantly distributed scalar values in space which can be organized in a grid (see Section 2.2.3 for further details on data set organizations). This type of organization is very efficient with respect to storage and data traversal. Interactive rendering times using software ray tracing were therefore demonstrated as early as 1998 [Parker99b]. While this was achieved using a supercomputer, later implementations relied on a cluster of consumer PCs or even a single PC [Wald05]. Since regular data sets can be easily stored in texture memory of graphics boards, ray casting implementations exist even for GPUs [Krüger03, Hadwiger05, Stegmaier05].

The major part of this chapter will, however, cover interactive rendering of volumetric data using implicit kd-trees. This was first introduced for iso-surface rendering allowing for several frames per second (fps), e.g. rendering the *Visible Female* data set [VHP] without any approximations [Wald05]. The latest improvement accelerated loading and rendering of massive data sets [Friedrich07] is covered in Section 4.4. A subsequent section shows, that this approach is not necessarily limited to iso-surfaces. Kd-trees may also be employed to accumulate values along the viewing ray allowing for semi-transparent rendering [Marmitt06a]. Advantages are achieved by skipping empty and homogeneous regions on high tree levels (see Section 4.5).

This chapter will then be closed with conclusions (Section 4.6), contributions (Section 4.7), and future work (Section 4.8) before extending *implicit kd-trees* for time-dependent rendering or applying it to other areas in Chapter 5. Before discussing the implicit kd-tree and its optimization, the following section continues first with an overview of other ray tracing implementations.

4.1 Related Work

The most obvious way for rendering regular volume data is an incremental grid traverser [Amanatides87], i.e. all cells along a ray are investigated whether they contribute to the final image or not. This brute force approach can be optimized in many ways as the following section will show. The next Section 4.1.2 covers non-hierarchical acceleration structures, e.g. *shell structures* and *proximity clouds*. Afterwards, hierarchical acceleration structures like octrees or multi-level grids are discussed (Section 4.1.3) before continuing with a discussion of the properties of *kd-trees*.

4.1.1 CPU and GPU Hardware Acceleration

The simplest scheme is the skipping of pixels. In that way Levoy [Levoy90b] traverses the initial image with interleaved pixels and interpolates in between. Subsequent images are adaptively refined. Lakare and Kaufman [Lakare04] shoot sample (called *detector*) rays into the scene not only accumulating the interpolated scalar values along the ray, but also tracking the first non-empty cell. *Leap* rays start from this position and traverse the volume. Lakare and Kaufman report a performance gain of up to 65% for various data sets. Interactive frame rates are restricted to a viewport of 256^2. It should be noted, that this approach does not work correctly in all cases. If the volume distance from the eye point exceeds a certain amount, rays begin to diverge too much, especially when using perspective rendering. Also, sampling artifacts may occur due to missed volumetric features.

Knittel's *UltraVis*-System [Knittel00] is a highly optimized grid traverser relying on empty space skipping and early ray termination. In addition, it provides perspective ray casting, trilinear interpolation, gradient shading, four light sources, and alpha blending. To achieve interactive frame-rates, a series of processor-specific optimizations are implemented. A spread-memory layout ensures that the volume data always stays in L1 cache. For the Pentium-III architecture with a four-way associative cache, four times the size of the volume data is allocated in main memory whereas the voxel data is stored only in the first quarter of each memory page used. Hence, voxels are only cached in the first quarter of each cache block and thus accessing voxel data almost never cause cache data replacements. The remaining parts can be filled with other frequently accessed data, e.g. lookup tables and other local data. For the ray-volume intersection test, conditional branches are replaced by SSE [Intel] masking operations allowing for 16 intersection tests in one loop, since packets of 4x4 rays traverse the volume simultaneously. Small data sets like the *Engine* or *MRbrain* can be rendered 2 to 10 fps on a single 500 MHz Pentium-III and 1 GB of main memory using a 256^2 viewport.

The *UltraVis*-System achieves high frame rates but was specifically suited for the Pentium-III processor. This makes it necessary to adapt this approach to every new generation of processors. In the meantime, GPUs appear to be an interesting alternative, since they became more and more flexible. A GPU volume ray caster have been proposed for several years but in most cases rely on multi-pass algorithms [Weiler03, Weiler04]. Recent advances show, on the other hand, that the latest generation of graphic boards allow the implementation of a single-pass ray caster on a GPU for regular grids.

One of the first implementations of volume ray casting using programmable graphics hardware was proposed by Krüger and Westermann [Krüger03].

They describe a GPU ray caster including typical ray casting optimizations, such as empty space skipping and early ray termination. While an additional octree allows for empty space skipping, the early ray termination is realized by using the *early z-test*, i.e. the value in the z-buffer is set to some maximum value preventing unnecessary invocations of the fragment shader. The rapid development of GPUs soon led to more sophisticated rendering frameworks.

Stegmaier et al. [Stegmaier05] and Hadwiger et al. [Hadwiger05] independently proposed frameworks for ray casting on the GPU in 2005. Stegmaiers et al. [Stegmaier05] approach uses dynamic branching and looping of todays GPUs. The regular grid is stored in a 3D texture but instead of employing texture mapping (see Section 3.5), the looping capabilities allow for implementing a grid traverser in the fragment program. The fragments are generated by rasterizing a polygon covering the screen space area of the volume's projected bounding box.

Stegmaier presented an iso-surface shader as an example by searching only for sign changes in the difference between the iso-value and the current and previous samples. A linear interpolation of the cell intersection (see Section 4.3.3) improves the rendering quality. The performance depends highly on the data set size and screen resolution, leading to 10 fps for 512^2 and 3 fps for 1024^2 for the *Engine* data set on an nVidia GeForce 6800 GT. The built-in parallelism of GPUs can be exploited further by clustering several graphic boards together [Müller06]. Quality enhancements can be achieved by implementing the Kubelka-Munk approach for tracking and visualizing reflectance and transmittance [Strengert06].

Hadwiger's et al. [Hadwiger05] approach is more advanced, since it uses multi-level grids already known from a software ray casting, but considers iso-surface rendering only. Like Parker et al. [Parker99b], a min/max value per brick (i.e. spatially grouped cells) allows for efficient culling of cells which do not contribute to the final iso-surface. A second advantage of using bricks is that they enable out-of-core rendering by swapping culled bricks from graphics memory to main memory. Whenever the iso-values changes, a range query checks all bricks, whether they possibly contain the iso-surface or not. Bricks not longer needed are replaced using a least recently used strategy.

Front and back faces of the volume's bounding box are rasterized to compute start and exit distances for the rays traversing the volumes. The linear interpolation is used iteratively as suggested by Neubauer et al. [Neubauer02] (further information can be found in Section 4.3.3). Shading occurs in image space by storing the computed intersection position in an off-screen pixel buffer. With a 512^2 viewport it is possible to achieve up to 20 fps even for mid-sized models and 10 fps for larger, out-of-core rendered data sets. Again, an nVidia GeForce 6800 GT was used.

However, the programming model, as well as the application interface of GPUs, is still tedious to use. For example, the number of loops in fragment programs is restricted to 256 at the current state. A ray caster therefore has to use nested loops for traversing the volume. Even then, secondary rays for advanced shading can hardly be implemented, since no recursion is available. First steps in porting ray tracing to GPUs have already been taken [Purcell02], but the future flexibility of graphics will determine of whether this will be a sufficient basis for allowing full featured ray tracing. CUDA [nvidia] seems here an interesting alternative but it still have to be shown, that it has advantages for volume rendering. The focus is therefore turned to software implementations with acceleration structures.

4.1.2 Non-hierarchical Acceleration Structures

Avila et al. [Avila92] present a complete visualization system based on ray casting called PARC (polygon assisted ray casting). Its components are a grid traverser similar to Amanatides and Woo [Amanatides87], combined with a polygonal visual hull enclosing non-empty scalar values within the regular grid. The GPU *z-buffer* is then employed to determine for each ray a near and far position for the grid traverser based on the intersection with the visual hull. This allows a ray to traverse non-empty cells only and hence to skip the unavoidable empty space surrounding each data set. Although this performance could be improved by a factor of 10, the rendering takes more than 5 seconds even for 256^3 data sets on a Silicon Graphics 240GTX with 256^2 screen resolution.

Shell structures, introduced by Udopa and Odhner [Udupa93], have a very similar concept. A shell is defined as a set of voxels in the neighborhood of the iso-surface boundary sharing the same range of opacity values. Voxels completely surrounded by high-opacity voxels are not stored in this set to save memory and are hence not fetched or traversed either. The saved memory is used for additional shading information, e.g. normals. Since rendering occurs in the shell domain, fewer voxels need to be fetched, and hence the rendering performance is 2 to 3 times faster compared to grid traversal. Although it would be possible to use shell structures for ray tracing, they choose a projection approach. The concept was later extended by Yagel [Yagel94].

Cohen and Sheffer [Cohen94] suggest *proximity clouds* to skip empty regions within a volume when using a grid traverser. *Proximity clouds* store the minimum distance to the next non-empty cell based on the city-block (i.e. rectangular shapes around each cell) or euclidian metric (i.e. circular shapes around each cell) in each cell. If a ray encounters a cell, the following n cells can be skipped, where n is the encoded minimum distance. The performance

increased by 30% compared to a grid traverser. Freund and Sloan [Freund97] adapted this idea in conjunction with transfer functions, i.e. the transfer function evaluation is directly encoded in the *proximity clouds* to skip non-interesting regions.

4.1.3 Hierarchical Acceleration Structures

Octrees are a fairly often used hierarchical structure. Levoy [Levoy90a] optimized his brute-force orthographic grid traverser by adding an octree for empty space skipping and early ray termination.

Wilhelms et al. [Wilhelms92] used an octree for iso-surface rendering of regular grids. To achieve this, the octree nodes are filled with the minimum and maximum scalar values found in the associated subtree. When traversing the octree, only those branches are taken that contain parts of the iso-surface. To save memory, eight adjacent cells are grouped together at the leaf level of the tree. When traversing a leaf node, for each of the contained cells a polygonal representation is rendered. This representation can be generated, e.g. with Marching Cubes [Lorensen87] in a pre-processing step.

Parker et al. [Parker99b] were the first to present an interactive iso-surface renderer for regular grids without extracting a polygonal representation explicitly. A multi-level grid enriched with min/max values allows for skipping large regions of the grid. On each level, all cells are grouped together in bricks (macro-cells) for improving the locality during traversal. Each cell can be easily accessed by storing its node index in a small table for the three-level hierarchy of the bricks. Once a cell with a possible intersection is found, a cubic polynomial is derived from trilinear interpolation (see Section 4.3.3), which is then solved by Schwarze's analytic inversion [Schwarze98]. Interactive rendering times still require 16 MIPS R10000 processors or more for a 512^2 viewport. Additional features like Phong shading [Phong75] and shadows decrease the performance further. Maximum intensity projection is also easy to implement by using a priority queue to track the cells or macro-cells with the maximal value, i.e. by using the maximum-value attached to each grid cell as priority.

A large drawback of this implementation was its restriction to rather expensive supercomputers. DeMarle et al. [DeMarle03] therefore extended the concept and adapted the algorithm for a cluster of consumer PCs. Here, a master node not only distributes image tiles to the cluster nodes, but each node is able to request a brick from all other nodes. This enables rendering of large data-sets like the *Richtmyer-Meshkov Instability* [Mirin99], which would not fit on a single PC. Using 32 PCs equipped with 1.7 GHz Dual-Xeon processors, the LLNL [Mirin99] can be rendered with up to 7 fps.

Grimm et al. [Grimm04] combined several techniques for rendering the *Visible Female* from the Visible Human Project [VHP] with up to 2.5 fps on a single 1.6 GHz Pentium-M, 1 GB main memory. It is a direct volume renderer supporting transfer functions but restricted to orthographic projection. The entire data is organized in bricks of 32^3 cells where the bricks themselves are stored in an octree. Bricks with homogeneous regions can be processed directly by using pre-integrated [Engel01] opacity tables. Inhomogeneous bricks are handled by a cell invisibility cache indicating whether a cell contributes to the final image with respect to the transfer function chosen.

Recently, Knoll et al. [Knoll06] used the octree not only for acceleration but also for compression. To achieve this, the volumetric grid itself is encoded directly into the octree by consolidating voxels with zero variance. The gained compression factor lies between 3 and 5 times, depending on the spread of isovalues within the data. The *Richtmyer-Meshkov Instability* [Mirin99] with 8 GB of data can then be rendered on a single 2.16 GHz Intel Core-Duo at near-interactive rates using a 512^2 viewport. Interactive frame rates are achieved on a 16-core NUMA 2.4 GHz Opteron workstation with up to 7 fps.

4.2 Background

Kd-trees in general as well as coherent ray tracing are covered first, before continuing with the description of the *implicit kd-tree*. This section contains theoretical background including pseudo-code for a generic kd-tree search. Section 4.2.2 describes the ideas behind coherent ray tracing and how this paradigm can be implemented on modern processors.

4.2.1 Kd-Trees

Kd-trees are a generalization of one-dimensional range trees, where a certain number needs to be retrieved within a given range. It is easy to see that a balanced binary search tree is suited best for this task. This guarantees that at each node within the tree, all children on the left are smaller than the split value and all values on the right are larger, or vice versa. Since the tree is required to be balanced, all its paths have length $O(\log n)$. Hence, the query time is $O(\log n)$, if a single point is going to be retrieved from the set. The build time is $O(n \log n)$, since the binary decision is $\log n$ for every point n within the set.

This idea can be extended to more than one dimension, since n-dimensional queries can be decomposed into n subsequent one-dimensional queries. The

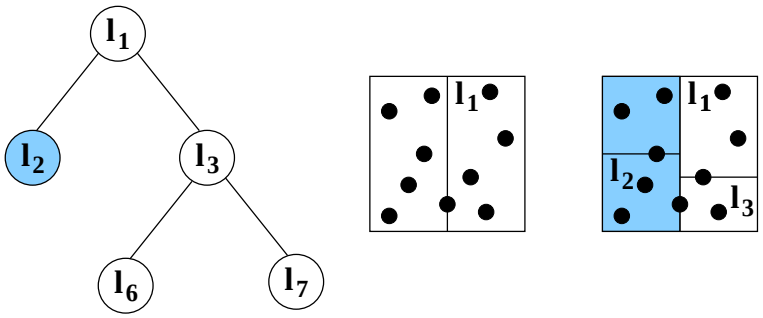


Figure 4.1: An example of a 2D kd-tree. The first split l_1 separates all 2D points with respect to the x-value, while l_2 and l_3 separate points with respect to the y-value. In this example it is assumed, that the x-value of the searched point is greater than l_1 and hence, the colored part of the point does not need to be investigated further.

solution here is to alternate the split between n dimensions, e.g. alternating between the x and y dimensions in the two-dimensional case.[1] An example for a 2D kd-tree is illustrated in Figure 4.1. By alternating between the x, y and z dimensions, the data structure needed for querying three-dimensional volumetric grids is obtained.

Theoretical build and query times do not change, since multi-dimensional points can be distinctly encoded in one dimension. The following pseudo-code demonstrates a recursive implementation for searching for a point within a one-dimensional kd-tree. As can be seen from Algorithm 5, the basic idea is quite simple. Note that the algorithm shows a split in the middle, which is generally not optimal for a given set of points. A more detailed discussion on kd-trees can be found in e.g. [de Berg00].

Kd-trees have been proven as a well-suited acceleration structure in theory [Havran01] and practice [Wald04a] for polygonal ray tracing. Since isosurface rendering (see Section 2.2.5.3) requires rapid location of cells associated with the defined iso-value, which is quite similar to a range search problem, kd-trees can be easily employed for this volume rendering technique. This is not only true for single, but also for packet ray tracing, as the following section shows.

[1] Originally, the name kd-tree stood for k-dimensional tree.

Algorithm 5 Search a Kd-Tree for a specific value.

```
seachKdTree(value, start, stop)
if stop − start = 1 then
  if node[split] = value then
    return true
  else
    return false
  end if
else
  split = (start + stop)/2
  if value < split then
    searchKdTree(value, start, split)
  else
    searchKdTree(value, split, stop)
  end if
end if
```

4.2.2 Coherent Ray Tracing

In Section 2.2.5.3, iso-surface rendering was defined as searching for the first (i.e. closest distance to the eye point) appearance of a user-defined value along a ray shot through the volumetric data set. This is similar to searching for the first triangle along the ray and calculating an intersection point. It seems therefore plausible to implement the same improvements which worked well for polygonal ray tracing. The ideas of coherent ray tracing are of special interest here.

Coherent ray tracing is based on the assumption that adjacent rays traverse more or less the same parts within the scene and hence the same kd-tree nodes. However, especially when using perspective projection, rays diverge with increasing distance and hence do not use, the same kd-tree nodes requiring them to be treated separately. Despite these shortcomings, coherent ray tracing showed a significant performance boost for polygonal ray tracing [Wald01a]. As will be shown later, this algorithm can be easily adapted to traversing regular grids. This demands several additional requirements to the implemented algorithm:

CPU-friendly algorithms. The implemented algorithm should favor modern CPUs and therefore try to use processor-specific extensions like SIMD in order to allow interactive frame rates.

Efficient data layout. Special care should to be taken with respect to the data layout in both cache and main memory when implementing data

structures. This requires for example that the nodes of a kd-tree be organized in a cache-aligned fashion.

The first of these two requirements is usually implemented by shooting and traversing packets of four rays through the kd-tree. To exploit coherence as much as possible, the rays are arranged in a 2×2 square which allows for simultaneous traversal when using the SSE implementation [Intel] of the x86 processor family[2]. The acceleration structure, in this case a kd-tree, should also simply extensible to this parallel or packet ray tracing. In contrast to other data structures, kd-trees demand only a binary decision per traversal step which is easy to implement. Grid-like data structures [Parker99b, DeMarle03] or octrees [Wilhelms92, Knoll06], require more decisions per traversal and are therefore impractical to combine with packet ray tracing.

4.3 Static Iso-surface Rendering

Rendering an implicit surface determined by a user-defined scalar value within a regular grid requires extracting that set of cells containing the searched iso-value. In other words, instead of accumulating all interpolated values along a ray, the problem is reduced to a binary decision whether a cell contains the iso-value and is hit or not (see Section 2.2.5). This decision is identical to the stated range search problem above so that the discussed kd-tree construction and traversal needs only minor modifications.

In general, the number of cells containing the iso-surface is rather small compared to the total amount of cells, but they are irregularly distributed in space. Since this set of (iso-surface) cells often defines the closure of the implicit surface, they are referred as *boundary cells*. After retrieving the cells containing the iso-surface, a cell intersection and typically a normal calculation are applied to render the final image. This is described in Sections 4.3.3 and 4.3.4. Section 4.4 will briefly sketch an optimized loading and rendering mechanism for massive data sets, before discussing a possible extension for semi-transparent rendering in Section 4.5.

4.3.1 The Implicit Kd-tree

As shown on the left side of Figure 4.2 a grid traverser have to check all cells along the ray until the cell containing the iso-surface is identified. A kd-tree

[2]Other processor families (e.g. PowerPC) offers similar but incompatible APIs for SIMD

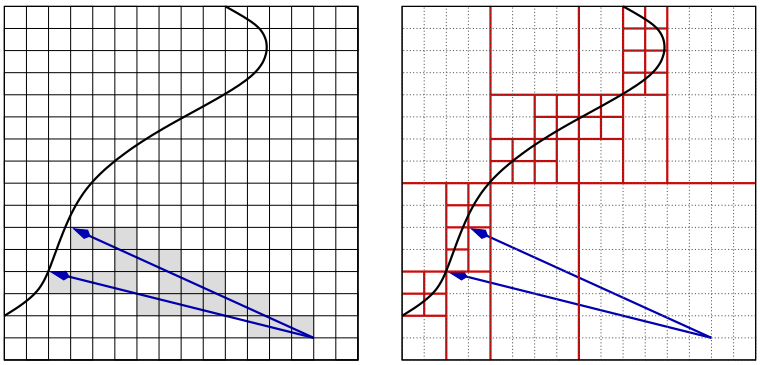


Figure 4.2: Left: *Traversing all cells along the rays until an iso-surface is hit.* Right: *A top down traversal in a min/max hierarchy can be used to quickly skip regions without boundary cells.*

provides a significantly faster alternative, since it requires less traversal steps, as the right side of Figure 4.2 illustrates.

This requires only two modifications of the kd-tree described in Section 4.2.1:

1. Each node within a kd-tree must store information regarding the iso-surfaces contained in the subtree represented by that node.

2. The traversal must be modified in such a way that each node is implicitly classified as to whether a subtree contains the searched iso-surface and is skipped if not.

A naïve approach would simply store a binary value in each node, i.e. *true* if a subtree contains the searched iso-value, *false* if not. Encoding the iso-surface *explicitly* in each node would however lead to the same problem, for which extraction was abandoned, since this would require the kd-tree to be built completely anew whenever the user changes the iso-value. The new structure should not have this shortcoming, thereby offering a significant advantage compared to extraction methods.

To achieve this, the kd-tree is built over *all* possible iso-surfaces at the same time by annotating each kd-tree node with information on what iso-surfaces it contains and performing the classification *implicitly* during traversal. One possible solution is to store the minimum and maximum values found in the grid region associated with a subtree. If the searched iso-value lies outside this range, the entire subtree can be skipped.

Since the tree still contains the entire data set, the iso-value can now be changed on the fly. The culling operation can furthermore be extended for

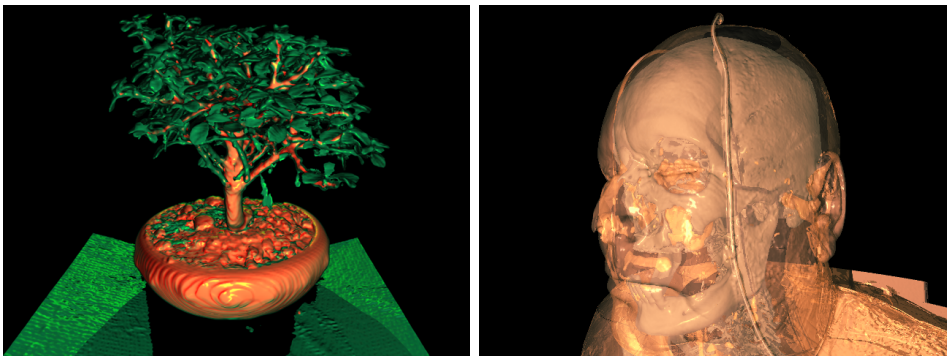


Figure 4.3: Implicitly culling non-contributing branches of the implicit kd-tree during traversal also enables rendering of multiple iso-surfaces at the same time. Left: *The bonsai tree, with a green iso-surface for the leaves, and a brown one for the trunk.* Right: *The* Visible Female*'s head, with bones visible through the semi-transparent skin surface.*

searching different iso-surfaces at the same time within one traversal operation. Data sets containing several iso-surfaces, like skin and bone surfaces for human data sets, can therefore be displayed simultaneously using blending operations (see Figure 4.3).

4.3.1.1 Tree Building

Building this kd-tree for iso-surface rendering is rather easy. In a recursive implementation, the root node contains the range of the entire 3D scalar field represented by its minimum and maximum scalar values. The volume is then split in a specific dimension, i.e. each of the two children of the root node represents now one half of the volume. The range of the scalar values is again the computed minimum and maximum scalar value found in the sub-volume corresponding to that node. As described in Section 4.2.1, subsequent splits could simply alternate between the x-, y-, and z-dimensions.

However, in this approach the dimension with the largest number of cells is always chosen as the splitting dimension (leading to the same scheme after some initial steps). The split always occurs in the middle so that the split plane coincides with the cell boundaries of the volume cells. This yields a one-to-one mapping between the volume's cells and the kd-tree nodes. In case of an odd number of cells in a dimension, the remaining cell is always assigned to either the left or the right child.

As previously stated, each node is annotated with the range of iso-surfaces contained within its subtree, which is simply the minimum and maximum

value of all scalar values found within the associated sub-volume. This can be computed recursively by first calculating the minimum and maximum values of a cell given by its eight corner values and propagating these values up to the root node. The minimum and maximum of one node is then simply the minimum and maximum, respectively of both children. Hence, each leaf node stores the minimum and maximum of its corner scalar values while each inner node stores the minimum and maximum of both children. This is similar to the approach described by Wilhelms et al. [Wilhelms92], except that they used an octree instead of a kd-tree, and for iso-surface extraction instead of ray traversal.

4.3.1.2 Tree Traversal

During each traversal step in a kd-tree, three traversal cases have to be distinguished. Given the distance from the ray origin to the splitting plane t_d and assuming that t_{near} and t_{far} represent the current ray segment in terms of the distance from the origin, the following three cases must be distinguished:

$t_{far} \leq t_d$**:** The current ray segment lies entirely in front of the splitting plane, i.e. only the front subtree needs to be traversed,

$t_{near} \geq t_d$**:** The current ray segment lies entirely behind the splitting plane, i.e. only the back subtree needs to be traversed,

$t_{near} < t_d < t_{far}$**:** The current ray segment overlaps the splitting plane and hence both children must be traversed by checking the front subtree first and putting the back subtree onto the stack.

Needless to say, t_{near} and t_{far} need be updated accordingly for the next traversal step. Figure 4.4 illustrates all three possibilities.

The missing link is now how to determine the ray segment defined by t_{near} and t_{far}. Since each node contains the minimum and maximum value (see Section 4.3.1.1), the searched iso-value is checked whether it lies between the node's minimum and maximum value, i.e.:

$$\rho_{min} \leq \rho_{iso} \leq \rho_{max}$$

where ρ_{iso} is the user-defined iso-value, ρ_{min} is the minimum value stored in the node, and ρ_{max} is the maximum value stored in the node. If this range check is only positive in the left child, this corresponds to the first case, i.e. only the front subtree is traversed. Inversely, if this range check is only positive in the right child of the current node, only the back subtree is

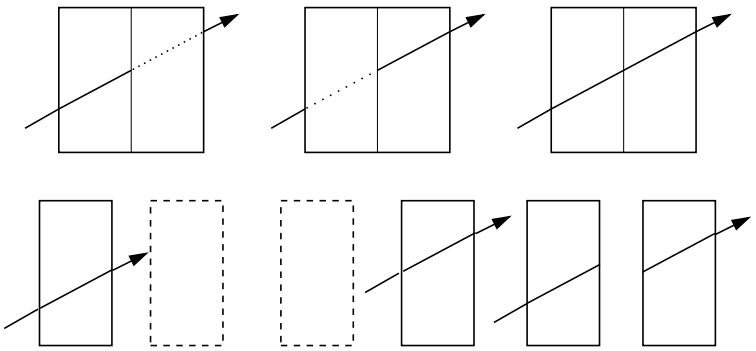


Figure 4.4: The three traversal cases are shown above. From left to right: *the ray segment is in front of the splitting plane, the ray segment is behind or overlays the splitting plane. This can be easily extended when shooting four rays in parallel, e.g. using the SSE extension [Intel].*

used for further traversal. If the iso-value is in both ranges, the left subtree is traversed and the right subtree is put on stack. In case that the remaining traversal of the left child fails, the first item is retrieved from the stack and the traversal continues.

4.3.1.3 Parallel SIMD Implementation

So far, only single ray traversal has been discussed without exploiting any type of coherence. It was, however, stated that the kd-tree was chosen because this acceleration structure works well with coherent ray tracing. In fact, the described traversal can be directly used for traversing four rays in parallel since one culling query can be applied to all four rays.

However, as mentioned above, this coherence can hardly be maintained until the leaf-level is reached. Highly diverging rays must be treated separately, which decreases the number of active rays in a packet and hence diminishes the performance gain expected using SIMD instructions. While the theoretical speedup is four, the practical speedup largely depends on the data set and perspective. The combination of a small screen resolution together with a large data set leads to volume cells covering only a single pixel. One would except therefore a relative small SIMD efficiency at least in this case. It should be noted, however, that the culling is subtrees is very efficient, i.e. the traversal decision is made on higher tree levels, where the rays are still coherent.

As can be concluded from Table 4.1, a high data set resolution, i.e. small cell size combined with a low screen resolution, is the worst case scenario in

Screen Res. Data Set	Data Res.	512x512 Single	512x512 Packet	Speed-up	1024x1024 Single	1024x1024 Packet	Speed-up
Aneurism	256^3	19.7	5.73	3.41	78.9	21.5	3.66
Bonsai	256^3	14.7	4.73	3.10	58.7	16.8	3.49
ML	32^3	7.93	2.17	3.64	31.7	8.33	3.81
ML	128^3	11.8	3.73	3.15	47.2	13.4	3.51
ML	512^3	15.5	6.39	2.42	61.9	2.71	2.95
Female	$512^2 * 1734$	5.57	2.82	1.97	22.3	9.56	2.33
" (zoom)	$512^2 * 1734$	8.33	2.08	3.99	33.3	8.36	3.99
LLNL	$2048^2 * 1920$	15.9	9.47	1.68	65.5	31.9	1.99
" (zoom)	$2048^2 * 1920$	13.3	3.32	4.00	53.2	13.3	4.00

Table 4.1: This table compares the number of traversal steps (in millions per second) for both single-ray and SIMD traversal with respect to varying scenes and screen resolutions. It turns out that the speed-up between single and packet ray is quite low for distant views of high resolution since nearly every ray within the packet has its own traversal path down the tree. However, screen resolutions or zoom-ins boost the traversal performance to the theoretical optimum.

which speed-up is around two. In other cases, the speedup increases to three or even the theoretical maximum of four.

4.3.2 Optimizations

While the previous section described building and traversal on an abstract level, it still remains unclear how to efficiently represent this kd-tree in memory. For the node information discussed above, this leads to twelve bytes per node: 4 bytes for plane dimension (30 bits) and orientation (2 bits), 4 bytes for a pointer to the right child and 2 bytes each for minimum and maximum value (assuming a 12-bit integer representation [Hounsfield80]). The structure of such a node, which is in the following referred as *large node*, is illustrated in Figure 4.5. For large integer or even floating point values, 16 bytes are required. Since leaf nodes do not require a pointer to the right child, this space can be used for encoding a reference to the associated grid cell. For regular grids, this reference is just the index value, since all scalar values are stored in an array.

Since each leaf node points to a cell, it is easy to compute the memory requirement for the entire tree. Let N be the number of cells or leaf nodes. Such a tree consists of exactly $N - 1$ inner nodes, which results in

Plane Dimension	Orientation	Pointer Right Child	Min	Pad.	Max	Pad.
30 Bits	*2 Bits*	*32 Bit (= 4 Bytes)*	*12 Bits*	*4 Bits*	*12 Bits*	*4 Bits*

Figure 4.5: This diagram illustrates each component of the large kd-tree node, assuming that the minimum and maximum values are stored as 12-bit integers [Hounsfield80]. In practice, the padding of 4 bits is added by using 16-bit short values.

$(2N - 1) \times 12$ bytes and hence occupy 12 times the size of the original data set in memory (which is $2N$). While for 32-bit values, the relative overhead is slightly better (7 times), it becomes worse for 8-bit values (20 times). The following four paragraphs describe five specific optimizations addressing memory consumption: reducing the node storage and number of nodes, relaxing the power-of-two constraint, discretized min/max values, and re-using parent min/max values.

4.3.2.1 Reducing Node Storage

A significant reduction can be achieved by assuming for a moment that the number of cells in each dimension is equal to a power of two, i.e. constructing a *balanced* kd-tree. No flag is needed indicating whether the current node is a leaf or an inner node. It is also easy to see, that all nodes in the same level l will use the same split orientation d_l. Hence, this information can be stored in a small lookup table equal to the size of the tree height with negligible memory consumption.

In a similar way, the position of the splitting plane does not have to be stored either. Let $R_{d,l}$ represent the number of cells to be split in dimension d in level l, i.e. level l splits $R_{x,l} \times R_{y,l} \times R_{z,l}$ cells. It follows immediately that there are at most $R_{d,l} - 1$ possible split locations. Since a split-in-the-middle is a necessary pre-requisite for balanced trees, it therefore suffices to simply save all possible split locations per level. Of course, additional overhead is added for the table lookup during traversal. Memory consumption, on the other hand, is reduced from $2N - 1$ to $2D - 1$ where N, the number of cells, is defined by $N = D^3$. Note that is possible to compute the splitting plane position during traversal, making the storage of this extra table obsolete. However, if the cell size varies within a dimension, i.e. the grid is anisotropic (see Section 2.2.3), it is costly to compute this position on the fly.

The storage cost can be reduced further by computing the pointers to both children during traversal. The children of a node at address n is simply $2n$ for the right child and $2n + 1$ for the left child. Not storing the pointer positions has another interesting side effect. This optimized version is inde-

Min	Pad.	Max	Pad.
12 Bits	*4 Bits*	*12 Bits*	*4 Bits*

Figure 4.6: This diagram illustrates each component of the small kd-tree node, assuming that the minimum and maximum values are stored as 12-bit integers [Hounsfield80]. The size of the kd-tree now depends only on the size of scalar values.

pendent of the size of the pointer and therefore can be easily ported to 64-bit architectures supporting even larger data sets (see Section 4.4). The memory overhead is already reduced from 12 to 4 bytes for 16-bit scalar values (see Figure 4.6). However, despite these enhancements, the needed memory for this *small node* scheme is still 4 times the size of the volume data.

4.3.2.2 Reducing the Number of Nodes

Using a balanced binary tree also means that half of the nodes are actually leaves. Not storing the leaves and instead computing a reference to them during traversal saves half the memory of the kd-tree. It turns out that this operation can be quite efficiently implemented in both C and SIMD code. Additionally, since these minimum and maximum computation only operate on a leaf level, they are far less common compared to computations in the inner node traversal. Additionally, due to efficient culling, all visited leaves require a cell intersection to compute the iso-surface.

Hence the memory overhead for this data structure can be reduced by another 50%, i.e. the storage cost for a kd-tree is now only twice as large as the data set size. Although a significant reduction was achieved by reducing the node size as well as the number of nodes, other approaches still require far less memory. Wilhelms et al. [Wilhelms92] octree adds only 50% of overhead, while Parkers et al. [Parker99b] hierarchical grid needs only 0.5% additional memory. On the other hand, only a fraction of the overall data contained in a kd-tree is actually accessed during a single traversal, making a factor of two quite tolerable.

4.3.2.3 Relaxing the power-of-two Constraint

So far the storage cost has been reduced to a factor of two but it was assumed that a balanced binary tree was used, i.e. the number of nodes in each level of the kd-tree was 2^i and hence the resolution of the data set was $2^i + 1$ values (i.e. 2^i cells) in each dimension. This is generally not true for volumetric

data sets. The simplest solution is to enlarge the data set to comply with this constraint by padding each dimension to a suitable resolution.

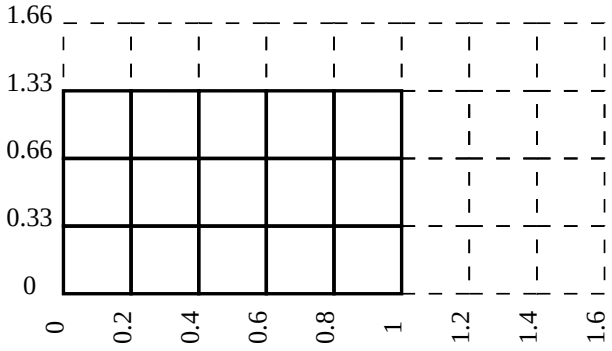


Figure 4.7: In this example, a grid consisting of 3×5 *cells is embedded in a virtual* 4×8 *grid with a balanced kd-tree. By Choosing appropriate split positions, it can be guaranteed that virtual cells never lie outside the scene bounds* $[0..1]^2$*, and thus will never be traversed by a ray. As a consequence, the nodes do not have to be stored and therefore consume no additional memory.*

The drawback of this optimization is the higher memory consumption if the the data set size does not obey the power-of-two constraint. The situation is especially poor for data sets with 2^i+1 cells in a certain dimension since nearly half of the cells will never be used. It is therefore better to distinguish between a *virtual* and a *real* grid size. The virtual grid exceeds the scene's original bounding box, and the kd-tree is built over this the virtual grid. Hence, it can be assured that all virtual nodes lie outside the real scene bounding box. Since the kd-tree traversal first clips the ray to that bounding box, rays will never be traversed outside this box, i.e. no ray ever touches any of the virtual nodes. From this observation, it follows, that such cells do not have to be stored. Using the same argument, leaf nodes of the kd-tree that point to virtual cells also do not have to be stored. To achieve this, the kd-tree is only built over the virtually padded volume, while the address computations and memory allocation are computed using the real grid resolution (see Figure 4.7). In other words, for a volumetric grid consisting of $R_x \times R_y \times R_z$ cells, a virtually padded volume $R'_x \times R'_y \times R'_z$ is constructed with $R'_{x,y,z} = min\{2^i | R_{x,y,z} \leq 2^i\}$ cells.

Groß et al.[Groß07] recently suggested to omit even these virtual nodes for the kd-tree. They use the fact, that for each kd-tree node the number of inner nodes is known advance. 3D boxes are used to calculate the split plane's position as well as the memory offset to the child node. This box is set

to the volume dimension when the traversal starts and during each traversal step this box is refined depending on the child's position in the tree. This increases the computational overhead only slightly but saves memory for the acceleration structure.

4.3.2.4 Discretizing min/max values

Wald et al. [Wald05] also suggested to discretize the minimum and maximum values stored in the nodes, although it was not investigated further. In a simple approach discretizing simply means that only the highest bits are stored in the kd-tree, e.g. the 8 highest bits of a data set with 16-bit scalar values. Since the leaf nodes are not stored, and therefore must be checked anyway by computing the minimum and maximum of the corner values on the fly, the introduced overhead should be negligible.

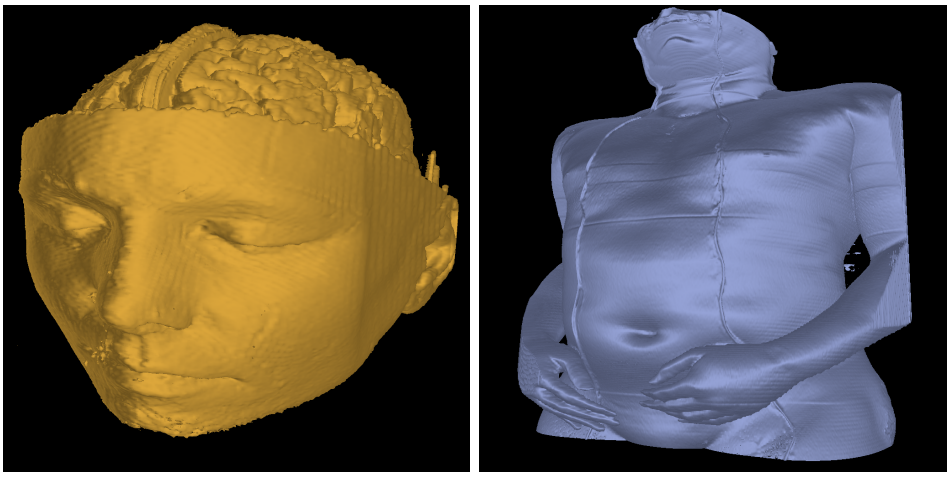


Figure 4.8: Left: *The MRbrain data set is typically used for testing twelve bit data sets (always 16 bits are stored) represented in Hounsfield units.* Right: *The Male Torso data set, also represented in Hounsfield units (see 2.2.3 and [Hounsfield80]).*

Therefore, only the simplest case is considered, i.e. discretizing the minimum and maximum values to 1 byte each. As an additional prerequisite, the compression scheme should be conservative, i.e. no node should be ignored during traversal which possibly contains the iso-surface. In the scheme described above, this can easily occur with the maximum value, since the lowest bits are removed. Rounding up the (discretized) maximum value by one avoids this problem, but still ignores the range problem. While it is easy

to discretize integer values, bit-shifting it is hardly appropriate for floating-point values. Even in the first case, this discretization is only optimal if the full range is actually used by the data set. In general this is not true, since the range might be for example between 200 and 2200 which would make the bit-shifting inefficient. As a simple solution, all input values are scaled and translated to the 8 bit range (0 to 255):

$$output = \frac{\rho_{iso} - \rho_{min}}{\rho_{max} - \rho_{min}} * 255$$

where ρ_{min} and ρ_{max} denote the minimum and maximum scalar values, respectively of the entire volume and ρ_{iso} is the original (non-discretize) scalar value to be converted to a value between 0 and 255. The same operation is of course applied to the iso-value given by the user. Using this equation, the range is optimally used. Table 4.2 shows the number of traversal steps and frames per second (fps) needed with and without compression for 16-bit integer and 32-bit floating point values. The results are intriguing (see Table 4.2) since one would expect a larger number of average traversal steps for discretized minimum and maximum values stored in the tree nodes. However, except for the *MRbrain* data set, the added traversal steps are negligible.

Data Set	Original		8-bit Discret.		%	%
	fps	#trav	fps	#trav	Δfps	Δ#trav
MRbrain (16 bit)	1.05	79.18	0.875	85.45	-16.67	7.92
SIMD	1.92	90.26	1.88	98.17	-2.08	8.76
Male-torso (16 bit)	1.03	63.86	1.07	64.16	+3.88	0.47
SIMD	1.54	91.33	1.57	91.93	+1.95	0.66
Drop (32 bit)	3.94	27.60	3.40	27.61	-13.71	0.04
SIMD	6.83	32.18	5.24	32.19	-23.28	0.03
Vortex (32 bit)	1.63	54.60	1.60	55.43	-1.84	1.52
SIMD	2.78	70.21	2.66	71.43	-4.32	1.74

*Table 4.2: Comparison of overall frame rates and corresponding average traversal steps for 16-bit integer (*MRbrain *and* Male-torso*) and for 32-bit floating-point (Drop and Vortex) data sets between non-discretized and discretized min/max values stored in the kd-tree. Similar average traversal counts reveal that the bottleneck lies probably in the different memory access patterns.*

This hypothesis is supported by the fact that volumes with a highly homogeneous distribution of scalar values significantly reduce the overall per-

formance (see performance for the *Drop* data set). Data sets with low homogeneity on the other hand may even lead to a performance gain as can be seen for the *Male-torso*.

4.3.2.5 Re-using the Parent's Min and Max Values

Another interesting storage reduction was recently presented by Groß et al. [Groß07]. The main observation is that maximum and minimum value are saved redundantly in either the left or the right child (see Section 4.3.1.1). By using two bits of the parent's node for indicating which of the children contains the identical minimum and maximum values the storage cost of the kd-tree is reduced to the storage cost of the data set (if no discretized min/max values are used).

4.3.2.6 Comparison of Performance

With the improvements explained in Sections 4.3.2.1 to 4.3.2.3, the memory overhead of the kd-tree is significantly reduced from a factor of twelve to a mere factor of two, independent of the data set's resolution. Table 4.3 shows a detailed comparison. The achieved reduction lies between a factor of three and ten. Note that large data sets, like the *Visible Female* [VHP] and the *Richtmyer-Meshkov Instability* [Mirin99] can not be rendered with the naïve implementation since the memory consumption is too high.

Scene	Data bits	Raw Data	Large Mem	Small			
				w Leaves		w/o Leaves	
				Mem	Ratio	Mem	Ratio
Bonsai	8	16MB	316MB	64MB	5	32MB	10
Aneurism	8	16MB	316MB	64MB	5	32MB	10
ML 32^3	16	65KB	680KB	220KB	3	110KB	6
ML 128^3	16	4MB	46MB	15MB	3	7.8MB	6
ML 512^3	16	256MB	3GB	1GB	3	509MB	6
Female	12(16)	900MB	–	3.4GB	–	1.7GB	–
LLNL	8	8GB	–	36GB	–	18GB	–

Table 4.3: Memory savings of the optimized ('small') versus the straightforward ('large') implementation. The 'small' representation can achieve memory reductions of up to a factor of ten. Larger data sets, like the Visible Female *and LLNL [Mirin99] data set, cannot be rendered at all with the naïve representation since the address bits in the 'large' node layout do not suffice for addressing such large data sets.*

The compressed *small* variant of the kd-tree needs more operations in each traversal step compared to the uncompressed *large* variant. This is especially true for tracking and updating all four indices (three for the dimensions and one for the tree level) of the current node as well as additional integer operations for computing the address of both children. The voxel indices must also be pushed and popped on the stack together with the near and far values to properly traverse the volume.

Scene	Single			SIMD		
	Large	Small	Overhead	Large	Small	Overhead
Aneurism	1.57	0.99	1.59	3.44	2.24	1.54
Bonsai	1.79	1.14	1.57	2.91	2.1	1.39
ML 32^3	2.47	1.47	1.68	4.92	3.41	1.44
ML 128^3	1.86	1.14	1.63	2.93	2.14	1.37
ML 512^3	1.30	0.91	1.43	1.62	1.24	1.31

Table 4.4: Comparing the performance of both the 'large' and the 'small' variant shows a significant overhead introduced by memory reduction. The performance loss might look high with 60%, but the memory reduction is necessary for rendering large data sets.

As can be seen in Table 4.4, all of these computations have a notable impact on the total rendering performance. Comparing the *small* and *large* variants shows an overhead of roughly 40 to 60 percent. Using SIMD the overhead is slightly reduced, since this code allows for amortizing address computation overhead over all rays in the packet. Remembering that the memory usage was reduced by a factor of ten, the performance loss of 50% seems quite tolerable. This is true in particular for lage data sets like the *Visible Female* [VHP] or the LLNL [Mirin99] data set, which originally could not be rendered with the employed hardware setup (see Section 4.3.5). Using the *small* variant is generally the better choice, except for very small data sets. Some more details on all tested data sets can be found in Section 4.3.5.

Including these optimizations, a new and efficient method was proposed to find the cell within a regular grid containing the iso-surface. The missing link to getting a rendered image is now the intersection with the iso-surface within a cell. Iso-surfaces are implicitly defined by their distributions of scalar values. The following section will discuss several intersection methods which were already described in [Marmitt04] that are suitable for interactive applications.

4.3.3 Iso-Surface Cell Intersection

While the previous sections covered in detail the detection of cells containing the iso-surface within a regular grid, it is now time to discuss the intersection of the iso-surface with an incident ray once such a cell is detected. Since the iso-surface is only implicitly defined by the discrete scalar values in space, the intersection point must be interpolated in some way. Section 2.2.4 already discussed the tensor-like product as one possible interpolation which will also be the basis for the following discussion.

Nevertheless, the following paragraphs distinguish between approximate and accurate intersection methods. The difference between these two methods is, that accurate methods are able to handle two or more intersections of the iso-surface with the incident ray within the cell. Approximate methods cannot detect such situations and therefore fail. One might think that this does not affect the rendering quality. As can be seen in Figure 4.9, this is not true. Approximate methods return *no* intersection point, if more than one exists, resulting in a black spot.

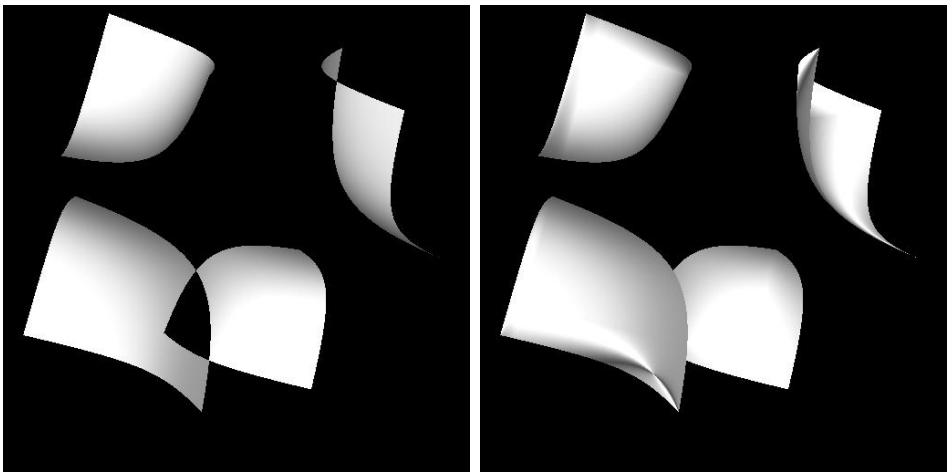


Figure 4.9: Sample scene for testing how approximate and accurate methods handle several iso-surfaces within one cell. Left: *Although the iterative linear approximation delivers smooth results, it fails if there exists two iso-surface intersection and returns false.* Right: *Accurate methods can handle this case correctly and return the first iso-surface along the ray.*

Accurate methods do not face this problem, as Figure 4.9 shows. Nevertheless, approximate methods are fast and deliver sufficiently good results

for some applications. Since approximative methods are also easy to implement on modern GPUs [Hadwiger05, Stegmaier05], they are still widely used. Accurate methods introduced so far [Parker99b] lack performance and are therefore not suitable for interactive applications compared to linear interpolation schemes.

4.3.3.1 Approximate Methods

The most simplistic algorithm assumes an intersection for every cell where the iso-value is within the range of scalar values at the vertices. One could take the arithmetic mean of the rays' entry and exit distances with respect to the cell found by the kd-tree traversal.

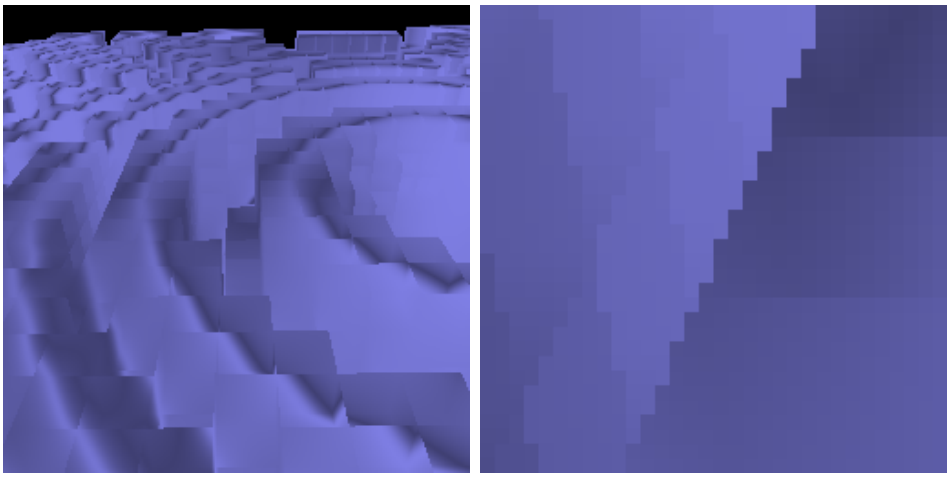


Figure 4.10: Left: *The* Marschner-Lobb *data set approximated with a midpoint cell intersection.* Right: *Zoomed view reveals that the iso-surface reconstruction is very rough.*

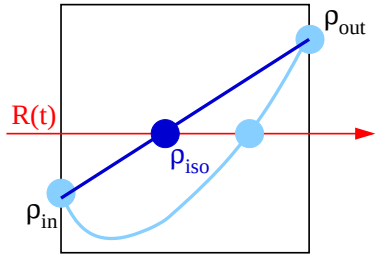
While this method is fast, it obviously leads to blocky artifacts when rendering cells (see Figure 4.10). Thus this method is only mentioned as a reference for performance comparisons.

Linear Interpolation Another simplistic, but more useful, approach assumes a linear function within a cell. Initially, scalar values at the entry and exit faces are computed by bilinear interpolation. This is reasonable since each face represents a quadrilateral primitive. By setting the interpolated entry value to $\rho_{in} = \rho(R(t_{in}))$ and the interpolated exit value to

$\rho_{out} = \rho(R(t_{out}))$, the iso-surface intersection can be computed with an additional linear interpolation using the user-defined iso-value ρ_{iso}, as outlined in Algorithm 6.

Algorithm 6 Linear Interpolation.

if $(\rho_{iso} < \rho_{in}) \vee (\rho_{iso} > \rho_{out})$ **then**
  return false
end if
return $t_{hit} = t_{in} + (t_{out} - t_{in})\frac{\rho_{iso}-\rho_{in}}{\rho_{out}-\rho_{in}}$



Though a bilinear interpolation for computing ρ_{in} and ρ_{out} suffices, it is generally faster to apply a trilinear interpolation since this avoids extracting the four scalar face values (which is, strictly speaking, also hardware-dependent). Another advantage is that trilinear interpolation works in favor of ray-parallel SIMD code, since it cannot be guaranteed that all rays enters and exits the cell at the same face.

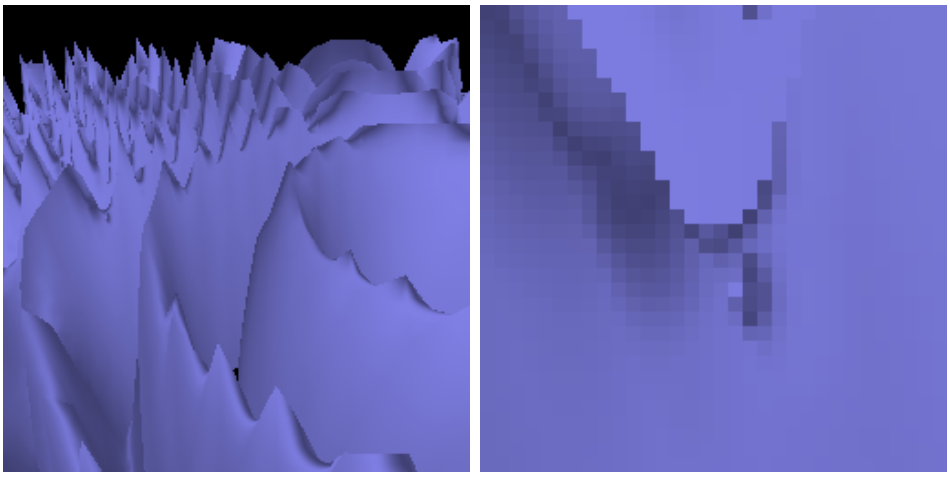


Figure 4.11: Left: *The* Marschner-Lobb *data set approximated with a linear cell intersection.* Right: *Close-up view shows a far better approximation compared to midpoint intersection.*

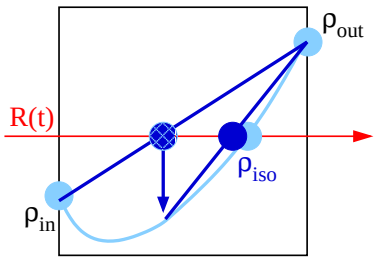
If the implicit (surface) function defined by the cell corner values are more or less planar, linear interpolation provides a good approximation. It

obviously fails, however, whenever this function has more than one root, in which case the entry and exit densities are either both larger than ρ_{iso}, or both smaller, and no intersection is detected (see Figure 4.9).

Repeated Linear Interpolation Neubauer et al. [Neubauer02] suggested a method that uses *repeated linear interpolation*. This method builds upon the linear interpolation just described, but refines the result iteratively. The value of at the linearly computed intersection point within the cell is trilinearly interpolated using the eight corner values. If this interpolated value is smaller than the user-defined value, the subsequent linear interpolation occurs between the entry point and the linearly calculated intersection point. Analogously, in the case of a larger interpolated value, the subsequent linear interpolation occurs between the linearly calculated intersection point and the exit point (see Algorithm 7). Typically, this iteration is applied a fixed number of times (two to three), even though an adaptive termination criterion is also possible.

Algorithm 7 Repeated Linear Interpolation

if $\rho_{iso} > \rho_{in} \ \wedge \ \rho_{iso} < \rho_{out}$ **then**
  return false
end if
for i=1..N **do**
  $t := t_{in} + (t_{out} - t_{in}) \frac{\rho_{iso} - \rho_{in}}{\rho_{out} - \rho_{in}}$
  if $\rho_{iso} > \rho_{in} \ \wedge \ \rho_{iso} < \rho_{out}$ **then**
    $t_{in} := t; \ \rho_{in} = \rho(R(t))$
  else
    $t_{out} := t; \ \rho_{out} = \rho(R(t))$
  end if
end for
return $t_{hit} := t_{in} + (t_{out} - t_{in}) \frac{\rho_{iso} - \rho_{in}}{\rho_{out} - \rho_{in}}$



Unfortunately this approach suffers from similar problems as the previous technique in that it sometimes fails to locate valid intersections. Nonetheless, in cases where it does correctly identify the intersection, the result is more accurate compared to simple linear interpolation. A special case occurs if there exist three intersections within a cell where two are located in the ray segment defined by the entry point and the initial linearly interpolated intersection point, i.e. the last intersection point is returned and not the first.

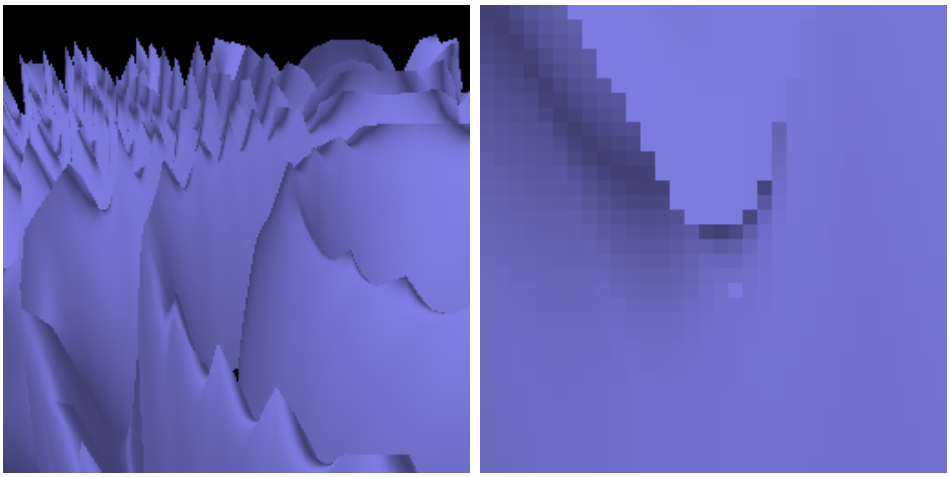


Figure 4.12: Left: *The* Marschner-Lobb *data set approximated with a repeated linear cell intersection.* Right: *Close-up view reveals only subtle differences between linear and repeated linear interpolation.*

4.3.3.2 Accurate Methods

Accurate methods must also interpolate the intersection value between the cell corners to calculate a meaningful iso-surface intersection. In contrast to approximate methods, it is guaranteed that the first intersection is returned no matter how many intersections exist along the ray. The following two methods both rely on solving a cubic polynomial and hence, take only the eight corner values of the retrieved cell into account, i.e. only C^0 continuity can be guaranteed.

This cubic polynomial is directly derived from the tensor product as with trilinear interpolation introduced in Section 2.2.4, on which the following considerations are based. Given a cell with eight scalar values ρ_{ijk} $(i, j, k \in \{0, 1\})$ at its eight vertices, the density ρ at any point $(u, v, w) \in [0, 1]^3$ can be computed by trilinear interpolation, i.e.

$$\rho(u, v, w) = \sum_{i,j,k \in \{0,1\}} u_i v_j w_k \rho_{ijk}, \tag{4.1}$$

where $u_0 = u$, $u_1 = 1 - u$, $v_0 = v$, etc. (see [Shirley05]). If the spatial location of this cell C is defined by $C = [x_0..x_1] \times [y_0..y_1] \times [z_0..z_1]$, then $\rho(p)$ of any three-dimensional point $p \in C$ can be computed by first transforming p to the unit coordinate system, yielding

$$p_0 = (u_0^p, v_0^p, w_0^p) = (\frac{x_1 - p_x}{x_1 - x_0}, \frac{y_1 - p_y}{y_1 - y_0}, \frac{z_1 - p_z}{z_1 - z_0}). \tag{4.2}$$

Using this notation, for a ray $R(t) = o + dt$ with origin o and direction d, which intersects the cell C in the interval t_{in} and t_{out}, the density $\rho(t) = \rho(R(t))$ for each point on the interval is defined as

$$\rho(t) = \sum_{i,j,k \in \{0,1\}} (u_i^o + t u_i^d)(v_j^o + t v_j^d)(w_k^o + t w_k^d). \tag{4.3}$$

Expanding this summation yields a cubic polynomial

$$\rho(t) = At^3 + Bt^2 + Ct + D, \tag{4.4}$$

whose coefficients (see [Parker99b, Shirley05]) are

$$\begin{aligned}
A &= \sum_{i,j,k} u_i^d v_j^d w_k^d \rho_{ijk}, \\
B &= \sum_{i,j,k} \left(u_i^o v_j^d w_k^d + u_i^d v_j^o w_k^d + u_i^d v_j^d w_k^o \right) \rho_{ijk}, \\
C &= \sum_{i,j,k} \left(u_i^d v_j^o w_k^o + u_i^o v_j^d w_k^o + u_i^o v_j^o w_k^d \right) \rho_{ijk}, \\
D &= \sum_{i,j,k} u_i^o v_j^o w_k^o \rho_{ijk}.
\end{aligned}$$

Finding the intersection of the ray with the implicitly defined iso-surface $\rho(t) = \rho_{iso}$ is then solved by determining the smallest $t \in [t_{in}, t_{out}]$ for which the polynomial

$$f(t) = \rho(t) - \rho_{iso}$$

is zero, i.e. the smallest root of f in the interval $[t_{in}, t_{out}]$.

Schwarze's Analytic Inversion This cubic polynomial can now be solved analytically using Schwarze's inversion algorithm [Schwarze98]. His method first checks for special cases (small coefficients A and/or B) where the polynomial can be approximated by a linear or quadratic representation. Note that even the special case of a quadratic polynomial already involves a costly square root operation.

The general case of a cubic polynomial is solved using Cardano's formula involving several cosines and even more costly inverse cosine operations. It

is furthermore necessary to compute *all* roots to locate the first one along the ray that is within the current cell.

Unfortunately, this algebraic solution is prone to numerical problems. The goal of interactive volume rendering unfortunately suggests the use of single-precision floating point values, which makes these issues even worse with respect to numerical stability. Consequently, it is difficult to tune this approach in order to completely avoid incorrectly computed intersections. While the Schwarze approach [Schwarze98] is often applied (e.g. [Parker99b]) and is mathematically one of the most accurate solutions for computing intersections in volume ray tracing, in practice it has too many drawbacks.

Iterative Root Finding Roughly speaking, the methods discussed so far are either slow and mostly correct, or fast and sometimes incorrect. Therefore, a new algorithm was derived [Marmitt04], that aims at being as fast as the Neubauer method, but is as correct as the Schwarze method. The new intersection method is based on the following two key observations:

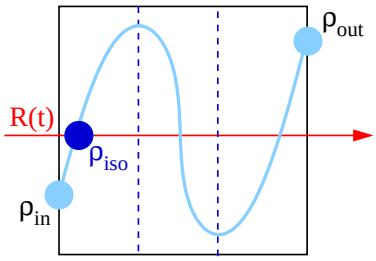
- Only the *first* intersection with the implicit function is of interest and there is no need to compute all intersections, as in the case of Schwarze's algorithm.
- Repeated linear interpolation *does* find the correct root fast and reliably *if* the start interval for the iteration contains *exactly* one root.

In this new approach, all roots are first isolated by computing the extrema of the polynomial. These two extrema then split the ray segment within the cell into at most three parts, where each segment is processed front-to-back by trilinearly interpolating the values at its start and end points given by entry and exit point and the polynomial extrema. Not that this is the only place, where a $sqrt()$ operation is necessary since a quadratic formula has to be solved. Once the interval is found, it is guaranteed that it (a) contains (exactly) one root (f is continuous, contains zero, and does not have extrema in that interval), (b) that the root lies in the interval $[t_{in}, t_{out}]$, and (c) it is the first root in this interval. Algorithm 8 shows a basic pseudo-code implementation.

After the interval is found, the root can be located as Neubauer described, i.e. by using a repeated linear interpolation or simply via recursive bisection. It is usually faster to apply a fixed number of two to three iterations than to apply an adaptive termination criterion and has only negligible impact on the accuracy. The known coefficients of the polynomial can be quickly and efficiently computed to calculate any data value along the ray, avoiding

Algorithm 8 Iterative Root Finding

{Find extrema: $f'(t) = 3At^2 + 2Bt + C$}
{// Quadratic formula needs sqrt()}
{Advance to intersecting ray segment}
for i=1..N **do**
  $t = t_{in} + (t_{out} - t_{in})\frac{-f_{in}}{f_{out}-f_{in}}$
  if $\rho_{iso} > \rho_{in} \ \wedge \ \rho_{iso} < \rho_{out}$ **then**
    $t_{in} := t; \ f_{in} = f(R(t))$
  else
    $t_{out} := t; \ f_{out} = f(R(t))$
  end if
end for
return $t_{hit} := t_{in} + (t_{out} - t_{in})\frac{-f_{in}}{f_{out}-f_{in}}$



further costly trilinear interpolations. This advantage is diminished when computing the coefficients and the extrema initially. As will be shown below, this approach is roughly faster by a factor of three than the Schwarze code, while providing the same guarantees on correctness and even better numerical stability. It is also well-suited for a data parallel SIMD implementation.

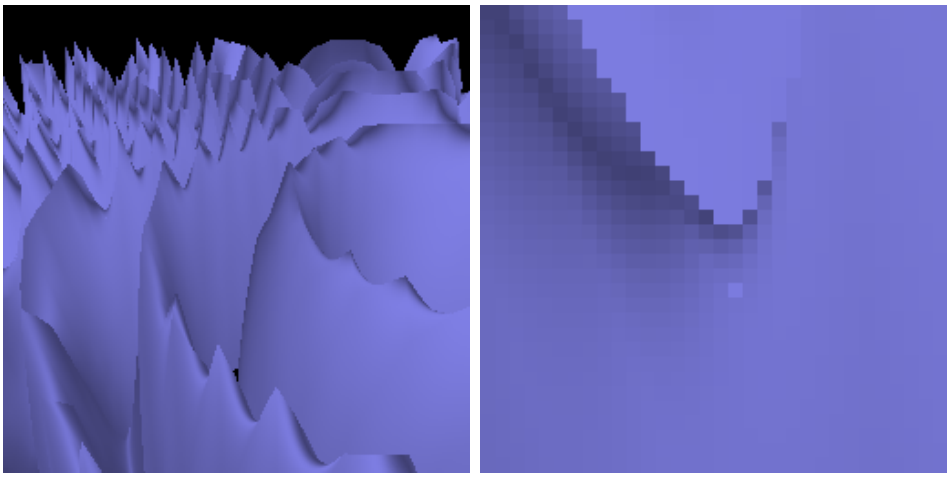


Figure 4.13: Left: The Marschner-Lobb *data set approximated with a iterative root-finding cell intersection.* Right: *Close-up view of the same data set demonstrating the achieved accuracy.*

4.3.3.3 Parallel SIMD Implementation

Wald et al. [Wald01a] already demonstrated the performance advantage of exploiting the SIMD instructions on today's processors using a data parallel approach, where the computations are performed on multiple rays in parallel. Modern SIMD instruction sets allow for operating on up to four floating point values in a single instruction, which can improve performance significantly, if the algorithm is SIMD friendly.

However, this is not the case for the Schwarze intersection due to its complex control flow for handling special cases and the evaluation of complex trigonometric functions. The new intersection technique is based on Neubauer's algorithm, but adds a SIMD-friendly computation of the polynomial's coefficients at the beginning. Despite the fact that SIMD involves masking operations and conditional assignments, the introduced overhead is negligible as it will be shown below.

The picture, however, is different when integrating the cell intersection algorithm into the rendering framework. Since the cell intersection necessarily occurs at the leaf level of the kd-tree, the ray coherence is rather low compared to the kd-tree traversal itself. One might think that this will do no harm to the performance, but SIMD instructions always introduce some overhead compared to the single-ray variant. A SIMD implementation, therefore, only makes sense if at least two rays are active. If only a single ray is active, any potential overhead of the SIMD implementation may even result in reduced overall performance.

Another disadvantage appears if not all rays intersect the iso-surface within the same cell. In this case, a large number of values must be stored to update the current hit position, including hit flag, hit distance, local cell coordinates, normal vector, etc. This leads to a total of eight values per ray which, in SIMD mode can only be handled using masking operations and conditional moves. This would lead to significant overhead together with low intersection coherence described above.

The code is therefore split into two phases: the first phase encapsules the computational core for a specific cell intersection, while the second phase simply stores the results. Since this computational core is completely implemented using SIMD, the computational density, as well as the floating point efficiency of intrinsic code, allow for an implementation which never slower than the single-ray code. It can thus be safely used even if a low degree of coherency is present at the cell level. The cost for the second phase, where the results are stored, can be reduced further by checking two special cases. If none or all four rays resulted an intersection, the amount of costly conditional moves is significantly reduced.

4.3.3.4 Comparison of Performance

The test system for performance measurements was equipped with an AMD Opteron Processor running at 1.8 GHz. Performance data has been collected by calling `gettimeofday()` before and after measurement. As the calling overhead of such an operating system function is likely to dominate the total execution time, the intersection kernel is being called several thousand times in between each timing call.

While this first setup was restricted to a single cell, the second setup incorporates the intersection test into a real-time iso-surface ray tracing system [Wald05]. As before, each kernel is called several thousand times in order to factor out any influence of the measurement procedure. The system was then run on several different real-world datasets, thereby yielding performance results that should apply to other real-world applications.

Method	Correctness	Bonsai	Aneurism	Engine
Midpoint	-	26.21	26.18	26.22
Linear	-	6.65	6.65	6.68
Neubauer	-	2.93	2.94	2.94
Schwarze	+	1.60	1.56	1.48
New	+	2.76	2.80	2.73

Table 4.5: Single ray intersection performance in million voxel intersections per second for real-world scenes. As for the synthetic data sets, the new method provides significantly better performance than approximate techniques while being as accurate as the much slower Schwarze method.

Using common real-world data sets[3], it can be seen, that the new algorithm shows a speedup factor between roughly 1.7 and 1.9 compared to the Schwarze algorithm that generates images of the same high quality (see Table 4.5). Note that all measurements are stated in million voxel intersections per second. The new algorithm yields similar performance as compared to the approximate Neubauer method, but always generates accurate images without artifacts.

For the SIMD versions basically the same measurements are applicable, but care must be taken such that four rays in parallel are used. Rather than simply dividing each result by four, it is better to divide by the number of active rays to compare this SIMD method and the single ray version directly. Table 4.6 shows the performance for the SIMD implementation. For the

[3]Measurements for synthetic data sets are provided in [Marmitt04]

Neubauer code, one can see a strong performance improvement by a factor between 2.3 and 2.6 compared to the sequential code. Neubauer's intersection seems particularly well-suited for some configurations that were not in the synthetic test suite [Marmitt04] included.

Method	Correctness	Bonsai	Aneurism	Engine
Midpoint	-	87.71	87.12	87.13
Linear	-	13.68	13.66	13.67
Neubauer	-	7.65	7.60	6.94
Schwarze	+	1.49	1.39	1.44
New	+	4.37	4.56	4.39

Table 4.6: SIMD intersection performance for real-world scenes. Again, the new algorithm is significantly faster than the Schwarze method, but computes the same correct results. All measurements are in million voxel intersections per second.

For the new algorithm the improvement is smaller due to the more complex control flow, but it still achieves about 60% higher performance through the use of SSE [Intel]. With the SSE implementation, the new accurate algorithm consistently achieve about 60% of the performance of the approximative Neubauer algorithm for the realistic data sets, with the additional advantage of correct intersection behavior.

4.3.3.5 Higher-Order Intersection Tests

The accurate methods discussed previously operate on one cell only to achieve the computational performance necessary for interactive applications. This implies, however, that there is no higher-order continuity, i.e. C^1 is not guaranteed at cell boundaries. In the following discussion an approach proposed by Rössl et al. [Rössl04] is briefly covered. It should be noted first that their method neither allows for interactive performance nor produces a complete C^1 continuity, but rather a pseudo-C^1 continuity. Neighboring cells are not taken into account directly, but the cell is subdivided into 24 congruent tetrahedra to achieve smoother iso-surface contours.

To compute the intersection of the incoming ray with the iso-surface, 65 Bernstein-Bezíer coefficients are calculated: 50 on the cell's faces, 14 on the median of each tetrahedron edge, and one at the center position of the cell. This is achieved by a repeated averaging pattern using 27 values interpolated in the interior of the actual cell and the 26 surrounding cells. This results in

ten coefficients per tetrahedron: four at the vertices and six at the median of each edge.

The ray iso-surface intersection may now be computed by solving quadratic equations along the ray for each tetrahedron. This quasi-interpolating spline results in a univariate piecewise quadratic polynomial, i.e. *quadratic super splines*. The ray intersection with each tetrahedron is computed in the following way. Given two intersection points of the ray with a tetrahedron, q_0 and q_1, as well as the arithmetic average $q = \frac{q_1+q_2}{2}$ together with ten Bezíer coefficients of the tetrahedron, the values w_1, w_2, and w are computed by applying the *de Casteljaus* algorithm at all three intersection points. The following equation then specifies the intersection point p of the ray with the iso-surface:

$$\alpha\tau^2 + \delta\beta\tau + \delta^2 w_1 = 0 \qquad , \tau \in [0, \delta] \tag{4.5}$$

where δ denotes the maximum hit interval, α is set to $\alpha = 2(w_1+w_2-2w)$, and β is set to $\beta = 4w - 3w_1 - w_2$.

Since this intersection test satisfies pseudo-C^1 continuity along the cell boundaries, the rendering quality is rather high. Even the shading normals can be directly computed from the polynomial pieces of the splines. Recall, however, that 65 coefficients are needed per cell. Even if some points are shared, the memory requirements are high compared to the previously discussed methods. For optimal performance, a careful tradeoff must be made between pre- and on-the-fly computation of all values needed for the intersection calculation. This was not further investigated by Rössl et al. [Rössl04] since their goal was to produce high-quality images. The reported performance of roughly 80 μs per cell seems therefore rather slow.

4.3.4 Shading and Gradient Calculation

After retrieving an iso-surface cell using a kd-tree and calculating the intersection with one of the methods previously discussed, shading and gradient calculation is the last part of a fully featured rendering system. Even for the simplest shading model, the directional light shader, it is necessary to compute the gradient at the intersection point to reveal surface-specific characteristics. This is shown in the following equation, where a light source is assumed at the viewer's eye position. The shaded color, here including an ambient term k_a, can then be computed with

$$I = (k_a + (1.0 - k_a) * N) \cdot D \tag{4.6}$$

where N denotes the normal (gradient) vector and some ambient value k_a between zero and one. A material color might be added by multiplying the result component-wise with its RGB-value. Note that this simple shader does not even need an intersection point, but a surface normal. More sophisticated shading models, like Phong [Phong75] shading, as well as refraction and reflection, may be added once an intersection point and the surface normal are known. Given a scalar field $f(x)$, the gradient vector is defined as

$$\nabla f(x) = \left(\frac{\delta f(x)}{\delta x}, \frac{\delta f(y)}{\delta y}, \frac{\delta f(z)}{\delta z} \right), \tag{4.7}$$

which should have unit length of used for illumination:

$$n(x) = \frac{\nabla f(x)}{\|\nabla f(x)\|}, \qquad \text{if } \|\nabla f(x)\| \neq 0. \tag{4.8}$$

The implicit definition of an iso-surface requires an estimate of this gradient. Considering that interactive frame rates are desired, finite differences [Engel06] supply the best tradeoff between speed and accuracy and will therefore be discussed here. Another advantage is that this can be easily extended for packet ray traversing. The basic idea is to differentiate a Taylor expansion of the function:

$$f(x_0 + h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} h^n. \tag{4.9}$$

Assuming that a one-dimensional function is solved for the first-order derivative, the *forward Taylor expansion* reads

$$f(x_0 + h) = f(x_0) + \frac{f'(x_0)}{1!} h + o(h^2) \tag{4.10}$$

and the *backward Taylor expansion* analogously:

$$f(x_0 - h) = f(x_0) - \frac{f'(x_0)}{1!} h + o(h^2), \tag{4.11}$$

where $o(h^2)$ denotes the approximation error, i.e. the remainder term of the power series. Subtracting Equation 4.11 from 4.10 results in the well-known central differences gradient approximation[4]:

[4] In order to maintain the approximation error function $o(h)$, it is necessary to calculate the Taylor expansion up to the third term. This leads to an error with an order of magnitude $o(h^2)$ and is thus of a higher order compared with forward and backward differences (see [Engel06] for more details).

$$f(x_0 + h) - f(x_0 - h) = 2f'(x_0)h + o(h^3), \tag{4.12}$$

solved for the first-order derivative:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + o(h^2) \tag{4.13}$$

Extending this approach to three dimensions is straightforward. Each of the three components of the gradient vector $\nabla f(x) + \nabla f(x, y, z)$ is estimated separately by a central difference resulting in

$$\nabla f(x) \approx \frac{1}{2h} \begin{pmatrix} f(x + h, y, z) - f(x - h, y, z) \\ f(x, y + h, z) - f(x, y - h, z) \\ f(x, y, z + h) - f(x, y, z - h) \end{pmatrix}, \tag{4.14}$$

Calculating the three-dimensional gradient requires six evaluations of the scalar function $f(x)$ and therefore, six trilinear interpolations. By normalizing Equation 4.14, with Equation 4.8, this vector can be used directly for the shading equation, e.g. for the directional light shader in Equation 4.6.

4.3.5 Results

This was the last step to implement a fully functional iso-surface renderer based on regular grids. It is now time to present performance measurements for the entire system, after the effect of SIMD (see Section 4.3.1.3) and other optimizations (see Section 4.3.2.6) were briefly discussed.

All performance measurements presented here are based on a *single* dual-1.8 GHz AMD Opteron 246 desktop PC with 2 GB RAM running Linux. The default resolution was set to 512^2 pixels. The complete code was compiled using the Intel C-compiler (*icc*) package with common optimizations.

Table 4.7 shows that this system achieves interactive frame rates for all tested scenes on a single PC, as long as the resolution is set to 512^2. In addition, ray tracing allows for a straightforward parallelization of the rendering process, so that it was possible to cluster five 1.8 GHz Dual-Opteron processors via Gigabit Ethernet. As expected, the frame rate scale approximately linearly with the employed number of rendering clients. The figures in Table 4.7 reveal that complex data sets can be rendered at up to 39 fps.

Table 4.7 also shows that the speedup gained by using SIMD for coherent ray tracing varies significantly depending on the average projected cell size. However, twice the performance can be expected, which is also confirmed when rendering time-dependent data sets with the concurrent tree update (see Section 5.1.2). In some cases, even the theoretical maximum of four could be achieved, making SIMD a worthwhile extension.

Scene	node type	Single PC			5-Node Cluster		
		Single	SIMD	Ratio	Single	SIMD	Ratio
Bonsai	large	3.4	5.2	1.5	16.2	24.6	1.5
Aneurism	large	3.0	6.2	2.0	14.6	29.8	2.0
ML 64^3	large	4.3	7.8	1.8	20.1	35.7	1.7
ML 512^3	small	1.2	2.3	1.8	6.1	11.3	1.8
Female	small	2.7	4.2	1.5	13.6	20.7	1.5
" (zoom)	small	2.3	7.9	3.5	11.2	39.1	3.5
LLNL	small	0.9	1.3	1.5	–	–	–
" (zoom)	small	1.6	5.4	3.9	7.6	28.7	3.8

Table 4.7: Overall rendering performance in fps when running the original framework in various setups including diffuse shading using both Dual-Opteron machines, as well as a five node Dual-Opteron cluster.

Another important aspect is the assumed sub-linear (i.e. logarithmic) scalability. When using a *hierarchical* data structure like the kd-tree, the advantages of polygonal ray tracing should be applicable for the iso-surface renderer also. This was verified by generating several resolutions of the synthetic *Marschner-Lobb* data set [Marschner94] measuring both the overall performance and the number of traversal steps. Plotting both measurements on a chart (see Figure 4.14) shows that this assumption is nearly true.

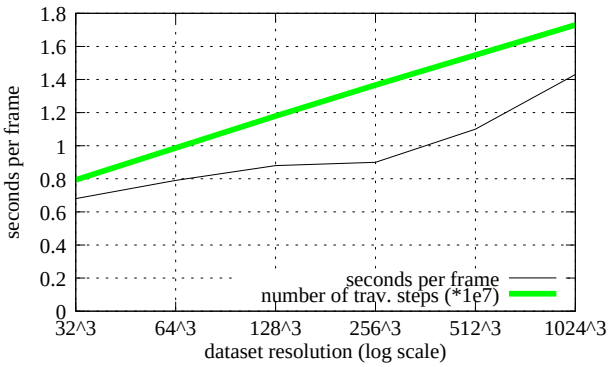


Figure 4.14: A scalability chart of the implicit kd-tree with increasing data set resolution. The resolutions of the synthetic Marschner-Lobb varies from 32^3 up to 1024^3. This logarithmic scalability leads to a performance drop by a mere factor of 2.1 despite the fact that the data size increases by 4.5 orders of magnitude.

The slight rise of the curve beyond 256^3 is most probably caused by caching effects for such large models. This is confirmed by the number of traversal steps (thick line), which does not show the same behavior. As for polygonal rendering tasks, this logarithmic scalability works best with extremely complex data sets. For example increasing the data set complexity from 32^3 (i.e. 3.2×10^3 cells) to 1024^3 (i.e. 10^9 cells) corresponds to 4.5 orders of magnitude in scene complexity, but leads only to a mere performance drop factor of 2.1.

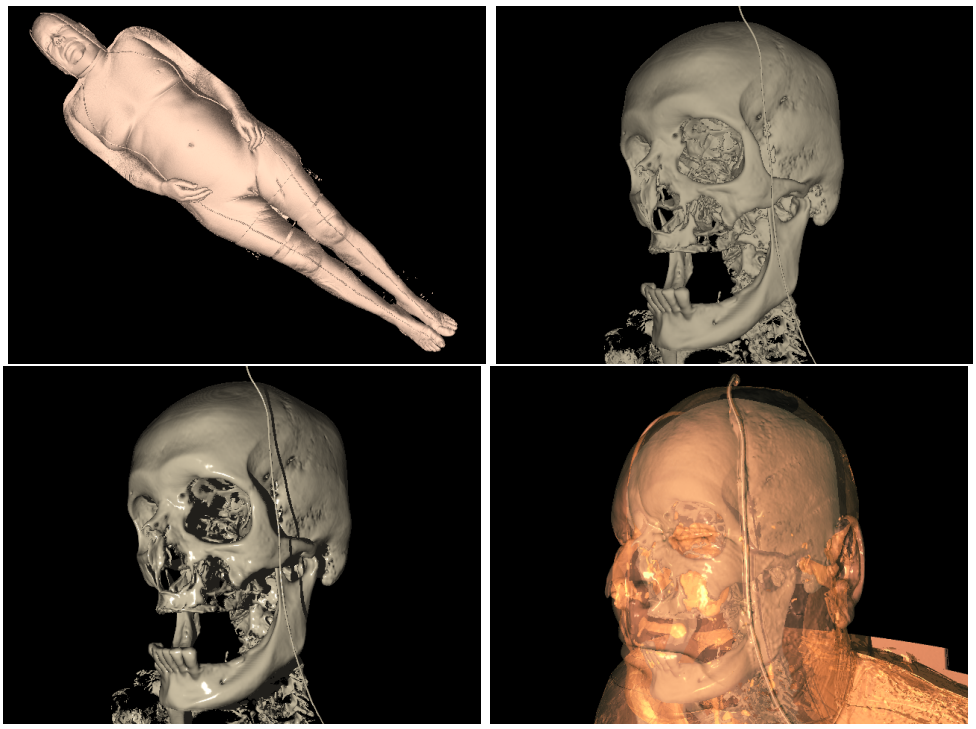


Figure 4.15: The Visible Female *(*$512 \times 512 \times 1920$*), rendered at* 640×480 *pixels on a single 1.8 GHz Dual-Opteron.* Upper Left: *Complete data showing the skin iso-surface (8.6 fps).* Upper Right: *Zoom-in of the head with the bone iso-surface (5 fps).* Lower Left: *Shadows added (4 fps).* Lower Right: *Shadows and semi-transparent skin added (0.8 fps).*

This last observation is important for the first example of the practical applications. This new framework already allows for an interactive exploration of complex iso-surfaces. Figure 4.15 shows that the *Visible Female* data set [VHP] can be rendered at 8.6, 5.0, and 4.0 fps at video resolution (640×480 pixels). Rendering the skin transparently introduces secondary rays, which were only implemented for single rays at that time. Still, approx-

imately one frame per second is achievable on a single PC. By distributing the rendering task among several processors (see Table 4.7), higher frame rates can be achieved.

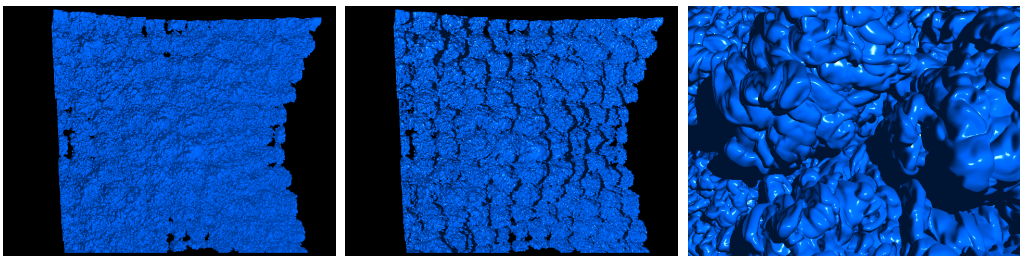


Figure 4.16: The LLNL data set, a $2048^2 \times 1920$ *simulation of a Richtmyer-Meshkov Instability.* Left: *Entire data set with directional light Shader.* Middle: *Same view, but with additional shadows.* Right: *Zoom-in of the surface to show the effect of shadows. Even such a complex data set of 24 GB of data can be rendered interactively on a single PC. At* 640×480 *pixels, these images render at 0.9, 0.3, and 1.1 fps on a single 1.8 GHz Dual-Opteron with 6 GB RAM.*

By combining this with an out-of-core rendering approach developed for ray tracing [Wald04b], it is even possible to render large data sets like the *Richtmyer-Meshkov Instability* from the Lawrence Livermore National Lab (LLNL) [Mirin99] with a resolution of $2048^2 \times 1920$ cells for each time step (see Figure 4.16). The usual iso-surface extraction is not really helpful, since this would lead to approximately 470 million triangles, which cannot easily be rendered using available graphics hardware. Of course, advantages can be taken from the fact that the implicit kd-tree scales logarithmically. However, due to the large input data set, this rendering consumes 24 GB of memory even for the *small* variant (8 GB data set + 16 GB kd-tree). By implementing a memory management unit for the Linux memory mapping mechanism, a single PC with 6 GB of memory is sufficient, to render this large data set at 2.1 fps when zooming in. With the addition of shadows, the performance drops to 0.3 fps or 1.1 fps depending on the chosen view.

4.4 Massive Iso-surface Rendering

In a joint work with Heiko Friedrich [Friedrich07], the latest extension of the implicit kd-tree for static iso-surface rendering aims at rendering large data sets on a single PC instantaneously. In a pre-processing step (see Figure 4.17),

several resolutions for the data set are computed using a Gaussian filter. A min/max kd-tree is then built for each level of detail (LOD) independently and are subsequently merged to a single tree valid for rendering all LOD levels. This merging ensures that the corresponding min/max intervals in each level are conservative, since the Gaussian filter shifts the range of values; thus, the kd-tree of the original volume is not necessary valid for all LOD volumes.

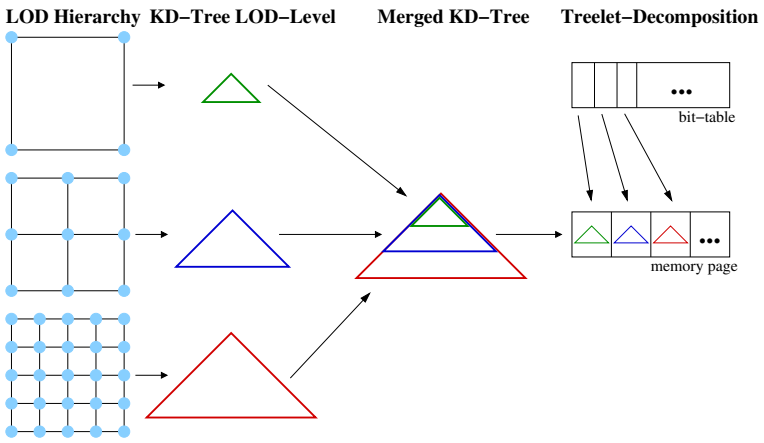


Figure 4.17: Illustration of the pre-processing pipeline from left to right. It starts with creating different LOD levels for the volume data sets, then builds a kd-tree for each level before merging all trees into a final single tree. The last step is to decompose this kd-tree again into subtrees (called treelets*) of a certain height and store then on the hard drive. Each* treelet *may also have two children, if it is not a leaf. During rendering, a bit table tracks the* treelets *currently available in the system's main memory.*

4.4.1 Treelet Construction and Traversal

This kd-tree is finally decomposed into so-called *treelets*. A *treelet* is defined as a subtree of the implicit kd-tree with a fixed height, consisting of an associated block of LOD voxels or the original data set, an *id* identifying the *treelet*, and the number of voxels for each dimension. This fixed height always corresponds to the number of chosen LOD levels. Except for the root *treelet*, all subtrees have the same size, based upon the page size of the Linux memory management (4096 bytes). Smaller *treelet* sizes are either padded to the page size, or multiple *treelets* are stored in one page if possible.

Additionally, the quantized value (half of the original size) of neighboring voxels need to be stored in a *treelet*, too. This enables the computation of surface normals of all but the quantized voxels within the treelet. As pointed out in Section 4.3.4, gradient calculations require the access of adjacent voxels since they are estimated using central differences.

Kd-tree traversal is similar to the original implementation previously described, except that a second level with *treelets* is added. Since all *treelets* have a fixed height, a conditional check of the leaf nodes can be removed from the innermost loop reducing the traversal cost considerably.

Treelet fetching occurs independently from rendering using a memory management unit (MMU) introduced in [Wald04b]. A loader thread loads the required *treelets* determined by the current iso-value so that the loading process does not stall the render threads. The loader sweeps over all *treelets*, which are stored in a breadth-first manner, and checks the range of iso-values stored for each *treelet*. Both children of a *treelet* are loaded if the searched iso-value is within this range. Each newly loaded *treelet* is marked as present in a global bit table, while discarded *treelets* are unmarked. The algorithm needs to know at LOD $i+1$ which *treelets* have been fetched in level i. This bit list is necessary since the algorithm needs to know at LOD $i+1$ which *treelets* have been fetched in level i, thus avoiding accessing unneeded data on the hard drive.

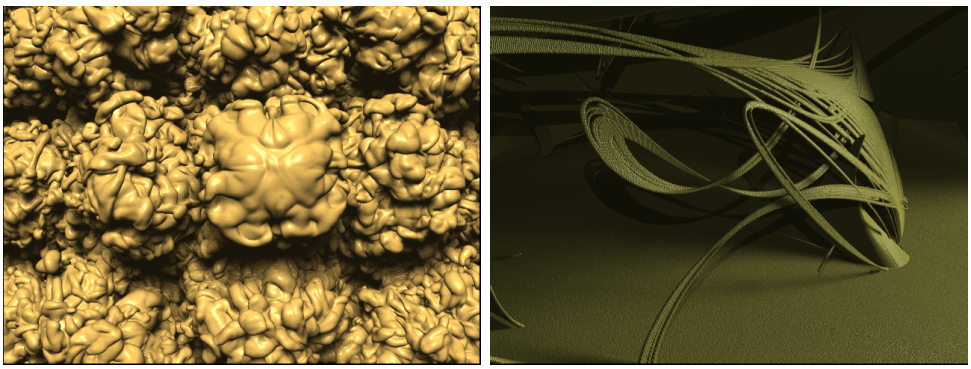


Figure 4.18: Example images of the tested scenes: LLNL and strange attractor rendered with soft shadows and Phong shading. Performance varies between 1.2 and 1.9 fps using video resolution and a single ray traversal.

Decomposing the LOD data and the *implicit kd-tree* into a *treelet* hierarchy is always a trade-off, since increasing *treelet* heights reduces the number of levels in the *treelet* hierarchy but increases the required disc space. It turns out, that a *treelet* height of nine seems to be a good choice taking storage

costs of the working set and the frame-rate into consideration. For example, the LLNL data set [Mirin99] can be rendered at 2.5 fps with a $640x480$ pixel viewport (see Figure 4.18).

4.4.2 Results

The original system [Wald05] needs 18 minutes to load all relevant data. The extension for massive data sets discussed here not only reduces this loading time to five minutes. Additionally the out-of-core mechanism was simplified and a conditional in the innermost traversal loop removed. Both enhancements nearly double the rendering performance. Even the memory requirement is reduced from 24 GB to 6.1 GB, as shown in Table 4.8.

Scene	Iso	Loading Time (Min)	RAM Space (GB)	FPS Diffuse	FPS Phong
LLNL (zoom)	16	5	6.1	2.7	1.3
LLNL (overview)	16	5	6.1	2.3	1.2
Attr (zoom)	25	4	2.1	3.1	1.6
Attr (overview)	25	4	2.1	3.5	1.9

Table 4.8: Overall rendering performance for the LLNL and the strange attractor. As the results show, independent from the view-point interactive frame rates can be achieved.

The overall performance was not only measured with the LLNL data set, but also with a synthetically generated strange attractor with a 2048^3 grid resolution using 8-bit scalar values. The testing system was equipped with a dual-core Opteron 275 (2.2 GHz PC with 8 GB main memory). Table 4.8 shows that interactive frame rates are possible in all cases, though the performance decreases by 20% when zooming in.

4.5 Static Semi-transparent Rendering

As explained in Section 2.2.5, iso-surface rendering is a special case of the volume rendering integral, i.e. the transfer function evaluates exactly one value as non-zero. Supporting an emission-absorption model seems therefore at first implausible for the kd-tree since every cell along the ray contributes to its final pixel value.

In general, a grid traverser [Amanatides87] is better suited for this task, since the next cell along the ray path can be faster detected. On the other

hand, it is impossible to skip certain regions using a grid traverser, which works in favor of a hierarchical acceleration structure. For regular grids, two types of regions can be identified which might possibly be skipped.

4.5.1 Adapting and Extending the Implicit Kd-tree

Due to the structured organization of regular grids, many scalar value locations are simply filled with zeros (and allowing positive values only as data). This is because the shape of the scanned object typically does not fit the box defined by the grid boundaries. If such regions could be identified, this would enhance the rendering performance. Using the kd-tree introduced in Section 4.3.1, such regions can be identified directly by testing

$$\rho_{min} = 0 \ \wedge \ \rho_{max} = 0$$

in each traversal step. If this criterion evaluates to true, the ray segment can be skipped completely. The performance gain of course depends on the height of the tree level in which such empty nodes can be found. Nevertheless, the *Engine* data can be rendered twice as fast keeping the other parameters the same (i.e. viewport resolution, eye point, and view direction).

However, a far better performance boost can be achieved when considering homogeneous regions, i.e. regions where the scalar values of all cells are equal or span only a small range. The idea suggested here is similar to Danskin and Hanrahan [Danskin92], where a pyramidal approximation was used to identify homogeneous regions. Freund and Sloan [Freund97] later adapted *proximity clouds* [Cohen94] (see Section 4.1) for handling homogeneous regions. Here, the implicit kd-tree can be used for identifying such regions by checking:

$$\rho_{min} = \rho_{max}.$$

The contribution for the corresponding ray segment is then set either to the value stored ρ_{min} or ρ_{max}. Since regions are in most cases not totally homogenous, it is often better to tolerate deviations up to a threshold ϵ:

$$\rho_{max} - \rho_{min} < \epsilon.$$

In other words, to identify homogeneous regions, the difference of the minimum and maximum values of the node can be checked against a user-defined ϵ-value. The contribution of such an identified region is now the only unknown left. One might think of just taking the arithmetic average, i.e. $\frac{\rho_{max}+\rho_{min}}{2}$, but this can result in severe rendering artifacts with increasing ϵ.

Empirical analysis showed, that it is better to compute the arithmetic mean of the entire region:

$$\rho_{avg} = \frac{1}{n} \sum_{i=0}^{n} s_i$$

with n denoting the number of scalar values s_i. This result is then stored in addition to the minimum and maximum values. To achieve this, the mean of the eight corner values for a cell is initially computed. The mean value of one node is then just the arithmetic mean of the mean values in both children. In other words, the propagation is identical to the minimum and maximum values discussed in Section 4.3.1.1.

4.5.2 Results

Skipping regions with a threshold introduces a visual error since the arithmetic mean is only a coarse approximation of the original data. Therefore it is left to the user when this coarse representation of the volume should be used. As already stated, the difference of the minimum and maximum values in a subtree can be compared against a user-defined threshold ϵ. Regions along a ray with more or less the same scalar values can be efficiently skipped in this way. Instead of checking each cell in the region, the arithmetic mean stored in each node is returned. The volume data itself is *not* compressed.

To check the introduced error, the same image was rendered with different thresholds. The well-known *Engine* data set ($256^2 \times 128$) consists of 8-bit integer scalar values allowing for a threshold range $t = [0, 255]$. As Figure 4.19 illustrates, choosing a threshold of $t = 64$ has only a negligible impact on the rendering quality while tripling the performance (see Table 4.9). Choosing $t = 128$ some artifacts, but many details are still visible. Further reduction ($t = 192$) leads to a blocky representation. However, regions with a low homogeneity are still clearly visible. These visual results are supported by an objective error metric, i.e. using the RMS-error metric defined as

$$RMS(I_0, I_1) = \sqrt{\frac{1}{n} \sum_{i=0}^{n} (p_{i,0} - p_{i,1})^2}$$

where I_0 and I_1 are the two images to be compared and $p_{i,0}$ and $p_{i,1}$ are pixels in I_0 and I_1 respectively. The calculated error (see Table 4.9) support, that the visual impression corresponds closely to this evaluation.

Extending the implicit kd-tree with an additional node value hence produces interactive results. The user can change the introduced error during

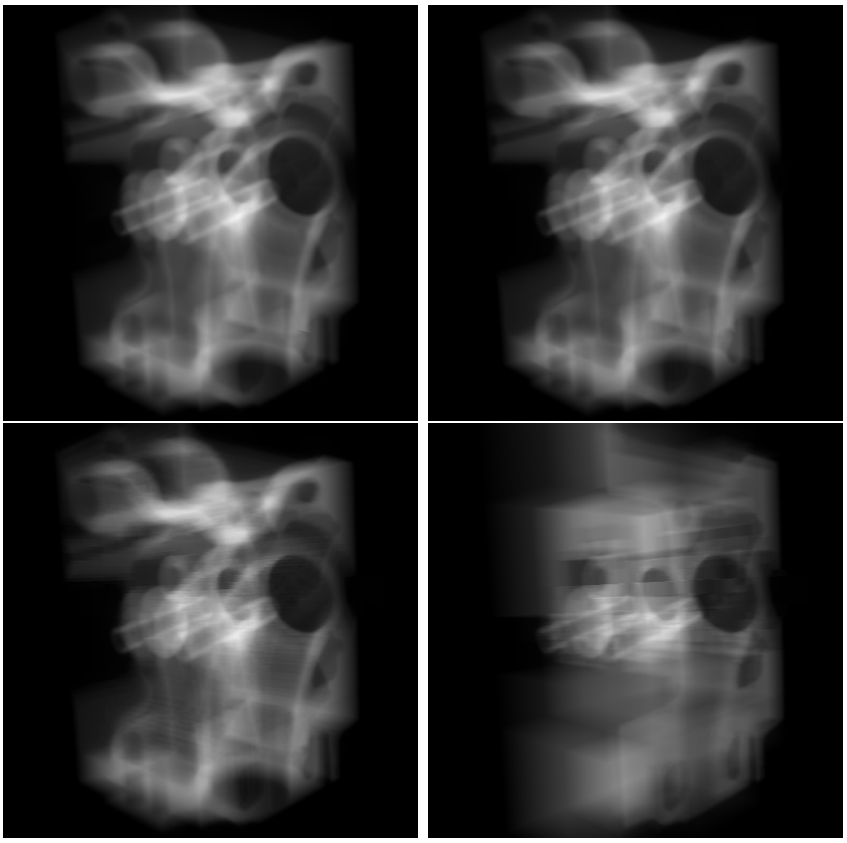


Figure 4.19: The engine data set rendered x-ray like with no threshold (upper-left), threshold 64 (upper-right), threshold 128 (lower- left) and threshold 192 (lower-right). Note that even for a fairly high threshold, regions with low homogeneity are still clearly visible.

rendering to make it suitable for the current visualization task. Future implementations might adapt this threshold automatically depending on the velocity of the camera movement, e.g. a large threshold during fast movements and a small threshold for minor or no camera movements.

It is even possible to switch between iso-surface and semi-transparent rendering, since the original node content was not abandoned. To save memory, the minimum and maximum values can be replaced with the difference, i.e. $\rho_{diff} = \rho_{max} - \rho_{min}$, leading to the same storage costs as for the iso-surface acceleration structure, i.e. two times the size of the volume.

It should be noted, however, that this can only be seen as a first step towards full support of semi-transparent rendering. For example, transfer functions are poorly supported right now, since the overall arithmetic mean

Threshold	RMS	fps	Speed-Up
0	0.0000	0.288	1.0
64	0.0063	0.946	3.3
128	0.0096	1.766	6.1
192	0.0785	4.022	14.0

Table 4.9: Different thresholds and associated errors introduced by rendering the coarse representation. The frame rate is measured for a 512^2 viewport on one dual-core Opteron with 2.0 GHz.

is obviously not a good approximation for transfer functions. One way to incorporate such functionality was already demonstrated by Subramanian and Fussel [Subramanian90], where regions not contributing to the current transfer function are marked in the kd-tree for skipping.

4.6 Conclusion

This chapter extensively discussed *implicit kd-trees* for rendering iso-surfaces defined over regular grids. The basic idea was to extend each kd-tree node with range values allowing the system to search different iso-values without rebuilding the entire kd-tree whenever the iso-value changes.

Kd-trees not only allow for an efficient retrieval of cells containing the searched iso-surface, but due to the binary decision during traversal, they are ideally suited for packet ray tracing, e.g. traversing the trees with four rays in parallel using SIMD operations.

This straightforward implementation was, however, memory-consuming compared to the original data set. A more efficient realization of the *implicit kd-tree* stores the range value only and additionally omits the storing of leaf cells. This leads to reduced storage cost, which is just twice as large as the original data set, but requires a balanced kd-tree. Virtual nodes help reduce the power-of-two constraint introduced with balanced kd-trees.

Although this memory optimizations reduced the performance by 50%, interactive frame rates of complex data sets even on a single PC are possible. This is especially true for the packet ray variant. Using the processor's SIMD extension enables the renderer to traverse four rays in parallel, leading to a speedup of two to four times compared to the single ray approach. An additional speedup can be achieved by distributing the image rendering process among several processor nodes. This allows for rendering data sets of several gigabytes, e.g. the *Richtmyer-Meshkov Instability* [Mirin99].

After retrieving the set of cells containing the iso-surface, it is necessary to apply an intersection test to the incident ray with each cell found. Some of the most relevant approximate and accurate methods were discussed here. Approximate intersection tests fail, if there is more than one intersection with the iso-surface within a cell. Up to now, the most often used accurate intersection method contained a complicated calculation for retrieving all possible intersections at once, although only the first intersection with a ray is relevant. The involved trigonometric equations adversely affect performance, especially since they have to be emulated in SIMD mode.

Iterative root finding was presented as an interesting alternative that combines the accuracy of the Schwarze intersection test with the speed of repeated linear interpolation. By checking the extrema values of the underlying third-order polynomial, an iterative bisection of the first region containing the iso-surface suffices for an accurate intersection calculation.

This hierarchical iso-surface rendering approach, which is based on the kd-tree together with a processor and cache-friendly implementation and cell intersection outperforms iso-surface ray tracing approaches previously published (e.g. [DeMarle03]). As demonstrated in Section 4.4, this is now true even for massive volumetric data sets. Using *treelets*, combined with an efficient memory management fetching scheme, large models can almost instantaneously be viewed and inspected. At the same time, the rendering performance is doubled compared to the original approach.

In all cases, no assumption needs to be made about the implicit surface. Storing the range values of corresponding subtree in each node enables full flexibility for the user. Since the iso-surface renderer is more or less a plug-in for a sophisticated rendering framework, lighting effects such as transparency, shadows, reflection, refraction, and even global illumination are instantly available. Regular volumes can also be combined with other primitives, e.g. triangles, free-form surfaces or other volumetric organizations. Interactive camera movements and iso-value changes are also possible.

However, many rendering visualization tasks require the complete evaluation of the volume rendering integral, or at least an emission-absorption model. At first, it seemed unwise to employ a hierarchical data structure since all cells along a ray contribute to the final pixel value and must therefore be checked anyway. Indeed, the a grid traverser [Amanatides87] offers due to its incremental approach a good rendering quality but lacks performance. Therefore the kd-tree nodes were enriched with an average value representing the arithmetic mean of the sub-volume. A homogeneity criterion was used to skip regions with only minor differences from the scalar value. Together with empty space skipping, this allows for interactive frame rates depending upon a user-chosen threshold. The idea is to set up this threshold automatically,

i.e. increasing the threshold for fast camera movements and reducing it for low or no camera movement at all. Another possibility is to to make the threshold depending upon the features of the data. It is noteworthy that even for a high threshold, non-homogeneous regions are still clearly visible so that a user can track regions of interest.

4.7 Contributions

The authors contributions to the topics discussed in this chapter are:

1. The author was involved implementing the *implicit kd-tree* [Wald05] and integrating the volume renderer into the OpenRT framework. This does not only include a full implementation of the iso-surface volume renderer for single and packet ray traversal, but also data set converting tools and performance measurements.

2. The author compared not only several ray iso-surface intersection methods in [Marmitt04] but presented also with iterative root finding a new fast and easy to implement intersection test which is as fast as a repeated linear interpolation while preserving the accuracy of Schwarze's interpolation.

3. The author contributed some minor implementation tasks to the massive iso-surface renderer [Friedrich07] based on *treelets*, e.g. a 3D gaussian filter and wrote a construction tool for generating large data sets.

4. The author completely developed and integrated an extension allowing for semi-transparent volume rendering in [Marmitt06a] using the *implicit kd-tree*.

4.8 Future Work

Although the static variant of the *implicit kd-tree* was optimized with respect to memory consumption, further reductions are possible. First steps have already been taken by reducing the node storage to minimum and maximum values and by quantizing these values. However, as seen for semi-transparent rendering, the nodes must been augmented with other important data to allow for efficient rendering.

One obvious idea is to increase the quantization even further. It is technically simple to use only four or two bits per stored value, although the trade-off between memory consumption and the number of traversal steps

must always be kept in mind. As already seen, higher quantization rates increase the number of traversal steps and hence, decrease the overall rendering performance. This effect was negligible, but may become significant for large data sets with a higher tree size since it not only increases the number of traversal steps, but also introduces costly bit shifting operations.

The first problem can be tackled by using *concatenated range values*, i.e. each level of the kd-tree stores a different bit position of the range values. It seems plausible to start at the tree root with the highest bit position. The second level would then store the bit for the second highest position, and so on. Having a tree with at least eight levels would hence lead to an 8-bit range value when the eighth level of the tree is reached. This follows from the observation, that higher tree levels contain larger ranges since they span large sub-volumes. Culling effects, therefore, often have a significant effect in the lower levels. Even when just 1 bit is used per level, the number of traversal steps should not increase significantly; however, increased bit shifting operations become a drawback.

Another issue is the order of split dimensions, which is currently chosen with respect to the largest dimension. Brauchle's Master Thesis [Brauchle06] showed here that a simple optimization scheme, i.e. maximizing the overlap of the ranges of both children's ranges, lead to significant speedup of approximately 20%. Details can be found in the upcoming Section 5.1.3. Although this idea was originally intended for improving the performance of the 4D kd-tree (discussed in Section 5.1.3), it can be adapted for the static renderer too. Other low-level optimizations require close attention to the exact caching behavior. This is particularly true for complex data sets.

In the same way, large volume data sets may also profit from this optimization. Furthermore, the massive iso-surface renderer should also support packet ray traversing and a lossless multi-resolution compression scheme in order to reduce the in-core and hard-disc memory footprint of this approach.

The proposed hierarchical semi-transparent renderer is in its current state not more than a proof-of-concept system. The current implementation does not support transfer functions well which are crucial for scientific visualization. Basically the transfer function needs to be 'convoluted' with the stored arithmetic mean to allow a meaningful usage of average values. The empty space skipping feature also needs an extension so that regions corresponding to the current transfer function are culled.

Grimm et al. [Grimm04] suggested a variety of techniques for rendering volumes on a consumer PC in software. Some of them could be adapted for this purpose, e.g. summed opacity tables allow for an efficient region skipping. Engels et al. [Engel01] pre-integrated volume rendering could be used to avoid the transfer function evaluation for every ray segment.

Even with these optimizations, highly non-homogeneous regions still force the traverser to locate the adjacent cell using the kd-tree. For such regions it seems far more effective to use an incremental approach, i.e. a grid traverser [Amanatides87]. A combination of both approaches, a kd-tree for skipping homogeneous regions and a grid-traverser for non-homogeneous regions, similar to [Levoy90a], should lead to a fast semi-transparent volume renderer supporting transfer functions with the above mentioned optimizations.

Finally, the CELL architecture seems to offer new opportunities for rendering regular volumes. A fast rendering engine for polygonal data was already presented in [Benthin06]. In the same way, the discussed hierarchical volume renderer should take advantage of this architecture.

Chapter 5

Dynamics and Other Applications

La rigidité est la rigueur des cuistres, qui ne sauraient jamais rien négliger. Mais qui ne néglige rien ne fait rien.

Gérard Genette

The last chapter extensively discussed the *implicit kd-tree* as a hierarchical acceleration structure for (massive) regular grids. Although best suited for iso-surface rendering, it was also extended to support semi-transparent rendering by skipping empty and homogeneous regions. However, time-dependent data sets have not been considered thus far since this would either require the system to rebuild the tree in each time step [Marmitt06a], or build a tree over the complete series of time steps [Brauchle06]. Both approaches are discussed in Section 5.1, including their advantages and shortcomings.

5.1 Time-dependent Volume Rendering

So far, the scalar value distribution was considered as static, i.e. the volumetric grid consisted of exactly one time step. However, there is a growing demand for rendering time-dependent volumetric data, such as simulations controlled by physical behavior, e.g. weather and climate forecasts, fluid dynamics, chemical reactions etc. Even medical applications sometimes require time-dependent simulations, e.g. a beating heart. This raises the question whether the discussed rendering framework can be extended to support rendering of time-dependent data.

Regarding the *implicit kd-tree*, there are mainly two options for this extension. As already stated, the *implicit kd-tree* contains the range of iso-values found in the associated region within a volume. Since it must be expected that the distribution, and hence the minimum and maximum (and average for semi-transparent rendering) change between individual time steps, these updates must be incorporated into the tree.

The first option leaves the node structure exactly as described previously but instead updates the kd-tree after each time step. This requires extending the existing framework with both a fast update and a synchronization mechanism. This concurrent update will be discussed in Section 5.1.2.

The other possibility leaves the kd-tree building within the pre-processing step, but augments the tree with a temporal dimension. Adding a temporal dimension to the existing three spatial dimensions is straightforward since the node structure for the spatial dimension can be kept as is. It turns out, however, that in this case the order of the dimensional split is crucial to achieve an acceptable performance. Details on this can be found in Section 5.1.3. Before describing the required extensions for both approaches,

the following section first covers related work in the area of time-dependent volume rendering of regular grids.

5.1.1 Related Work

Due to the high computational demands of time-dependent volume rendering, extensive research started only a couple of years ago. Shen et al. [Shen99] propose the so-called *Time-Space Partitioning* (TSP) tree for efficient culling of spatiotemporal homogeneous regions. TSP trees consist of an octree as a spatial hierarchy where each node is extended with a binary tree for traversing the temporal dimension. Hence, all nodes of this binary time tree associated with a TSP tree node represent the same subvolume in the spatial domain but with differing time span. Mean values are used for homogeneity acceleration where additional values represent temporal and spatial error metrics. This algorithm was later adapted to graphics hardware using a slicing algorithm [Ellsworth00].

Reinhard et al. [Reinhard02] extend the interactive volume ray tracing system introduced by Parker et al. [Parker99b] for time-dependent iso-surface rendering. The I/O bottleneck when loading the next time step is significantly reduced by storing only those voxel values that differ from the current time step, i.e. all subsequent time steps are represented as a list of scalar values associated with their spatial coordinates. During rendering, one processor reads the next time step and updates the volume in main memory while all others render the data set. Rendering with interactive frame rates need at leat 16 processors on an SGI Origin 2000.

Gao et al. [Gao04] exploit the temporal coherence between adjacent time steps by grouping cells with temporal and spatial coherence together in volume blocks. *Plenoptic Opacity Functions* (POFs) [Gao03] then encode the minimal occluding capability of a volume block from an arbitrary viewpoint. The POF as well as the temporal and spatial coherency information for each block is stored in an octree. They reported a render time of 1.4 seconds per time step for the *Richtmyer-Meshkov Instability* [Mirin99] on 32 2.4 GHz Xeon processors.

Younesy et al. [Younesy05] introduce *Differential Time-Histogram Table* (DTHT) to store changing voxel values between two adjacent time steps. This table has two dimensions, where the size of one dimension is equal to the number of time steps and the other depends on the range of scalar values stored in a bin. Each bin refers to the active set of iso-values, which need to be rendered in the current time step, and a set of differences between the active sets of adjacent bins. The active set of iso-values is a range determined by corner values of all cells possibly containing the iso-surface. For each time

step, this table updates the changed voxel values with respect of the next rendered time step. While the update performance was reported as more than 10 fps, rendering times were not reported.

5.1.2 Concurrent Tree Update

The approach discussed in the following paragraphs is based on the idea that a number of update nodes is employed to update the kd-tree for upcoming time steps concurrently, while the latest available tree is traversed by a number of render nodes. This makes it necessary to adapt the system discussed in Section 4.3 for shared-memory architectures since it is now vital to provide fast construction of the kd-tree, which is accessible to all render nodes. Note that in the original implementation [Wald05], each processor node built and used its own copy of the kd-tree due to static volume data sets.

Kd-tree building is accelerated by two observations. The original implementation built the kd-tree recursively. Although easy to implement, recursive functions cannot be efficiently translated by today's compilers. Neither stack usage nor data access is optimal. Additionally, construction was not distributed across several processors, which is difficult for recursive implementations anyway. By carefully designing an iterative tree update (see Section 5.1.2.1), parallel kd-tree updating (see Section 5.1.2.2) is enabled at the same time. It turns out that both optimizations suffice for small time-varying data sets up to 256^3 on the shared-memory system.

5.1.2.1 Replacing the recursive Implementation

The first step is therefore to replace the recursive build by an iterative update, which works in two steps. In the first step, the scalar values of a volume are fetched from the data set cell-wise (see Figure 5.1). For each cell, the necessary values for the tree leaves, e.g. minimum and maximum for iso-surface rendering (see Section 4.3), or a mean value for semi-transparent rendering (see Section 4.5) are computed and stored in the array used for the kd-tree. Care must be taken that access to the volume scalar values (stored as an array in memory) is as sequential as possible. Due to the one-dimensional memory access, this can only be achieved for one of the three spatial dimensions, usually the x-dimension. This improves the reading performance by approximately 20% compared to using the last split dimension for accessing the data.

The kd-tree nodes cannot be sequentially accessed since neighboring tree nodes in one level do not necessarily refer to adjacent volume cells. Since there are eight volume scalar values to read compared to two values to write

to the kd-tree array, giving preference to the volume read seems a better choice. Considering the fact that adjacent cells share four points, the number of fetches is reduced to four per processed cell (see Figure 5.1). Finally, since the small variant of the kd-tree was used (see Section 4.2.1), the leaf level of the tree is not stored either. Instead two adjacent leaf nodes in the last split dimension are merged directly, i.e. the node content is computed for twelve values (four values are shared and must not be counted twice).

A further acceleration can be achieved by bricking the volume data as suggested in [Parker99b, DeMarle03]. Bricking simply means that adjacent cells in all three dimensions are grouped together in memory so that they can be accessed within the same fetched cache line. Pre-computed access tables help keep the computational overhead for index calculation low [Parker99b]. Of course this heavily depends on the cache architecture. Since the volume is accessed on a cell level, bricking $2 \times 2 \times 2$ cells is a good choice because this corresponds exactly to a cell. Bricking gives an additional performance gain between 10% and 15% for the kd-tree update.[1]

Since the last level of the kd-tree is now filled, it is rather easy to propagate the computed minimum, maximum, and mean values level by level to the root node in a breadth-first manner. This is quite similar to the update mechanism used for dynamic hardware ray tracing, presented in joint work with Sven Woop [Woop06]. Note that in this step, it is not necessary to access the volume data itself since basic arithmetic operations suffice for propagating minimum, maximum and mean values rather quickly compared to the first step. Figure 5.1 illustrates this step. Empirical analysis showed that using a separate array (blue color) for intermediate caching is faster, which is believed to be caused by a better cache usage when propagating the values.

Until now this method has been three times faster than a recursive implementation (see Table 5.1 and 5.2), but it is still not fast enough for interactive tree updates. This is achieved by using another property of this implementation. The following sections will describe how to add multi-threading and synchronization to complete the rendering framework for time-dependent data.

5.1.2.2 Multithreading

As already mentioned, this approach can be used to distribute the update routine among several processor nodes. Each processor node can update a part of the tree and a master process can simply merge them together.

[1] Theoretically, a 2-2-n bricking, where n is the size of the z dimension is the optimum. However, this destroys adjacency with respect to the x and y dimensions completely and therefore does more harm than good.

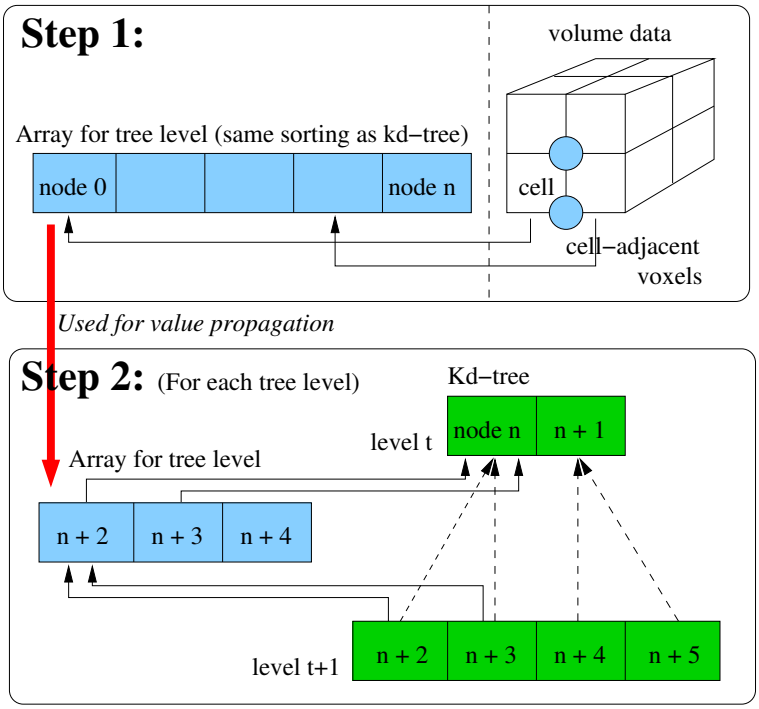


Figure 5.1: In the first step, all values are fetched sequentially from the data set and the information to be stored is sorted into the tree. The following step propagates these values bottom-up in a breadth-first manner. Since adjacent cells share four voxel, they do not have to be fetched twice, i.e. only four instead of eight values must be fetched per cell.

As an example, consider the case where the first splitting plane is in the x-dimension. Distributing the ranges of these two new subvolumes to the iterative update routine, it is easy to see that these two sub-trees can be built independently (see Figure 5.2). Only the first level needs a merge by the master.

In the same way, it is possible to update this kd-tree by four, eight, or more nodes. The only restriction is that there must be a correspondence between the subvolume represented in the entire tree and the update routine, i.e. the number of parallel processes must be equal to 2^n. Alternatively, virtual nodes described in Section 4.3.2.3 can be used. A split in the middle naturally leads to a good load-balancing between all threads since the performance is only dependent on the number of scalar values processed.

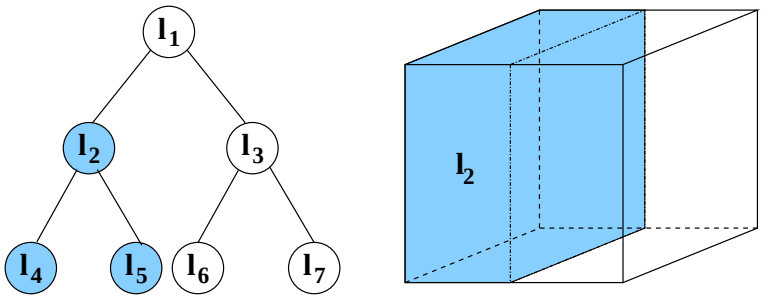


Figure 5.2: In this example, two processor cores update the complete tree. By assigning to each tree the subvolume corresponding the splitting plane, both subtrees can be independently built and merged together. At the same, a split in the middle guarantees time load balancing.

5.1.2.3 Update Performance Speedup

The pure performance of the described update routine was tested with the well-known *Marschner-Lobb* [Marschner94] in various resolutions. Here, the tree is updated every time with the same data set. Caching issues are negligible due to the size of the data. Despite the fact that the operating system completely handles the thread management, our experiments showed that the processing time scales linearly for nearly all cases (see Table 5.1). The test system was, however, equipped with 16 cores so that the decrease might not be linear. However, with 16 threads the decrease is no longer linear, since the maximum number of cores on the test system is reached. It is believed that memory bandwidth becomes a limiting factor in this case. Nevertheless, a kd-tree for a 512^3 volume can be built in approximately 1.5 seconds using 16 threads.

Data	rec.	iterative (# threads)				
size	1 thread	1	2	4	8	16
64^3	0.056	0.016	0.008	0.004	0.002	0.002
128^3	0.454	0.157	0.072	0.043	0.023	0.017
256^3	3.784	1.333	0.687	0.357	0.197	0.160
512^3	29.75	10.74	5.396	2.945	1.930	1.543

Table 5.1: Updating the iso-surface tree in our threaded version shows that the update time (in seconds) decreases linearly with the number of threads as long as the operating system can distribute the threads to free nodes.

Table 5.1 shows the performance in seconds for updating the acceleration structure for iso-surface rendering. Detailed analysis showed that most of the time is spent in the first part of the update routine where the data from the original volume is fetched. Over 80% of the total update time is spent in this part, while less than 20% is needed for bottom-up propagation. This is caused by the heavy volume read load (twelve reads versus two writes). The time for merging the trees for even 16 threads is negligible.

Evaluating our system with the tree structure for emission-absorption models also shows a linear decrease with the number of threads used. Compared to the iso-surface tree, the timings are approximately 17% higher (recursive implementation: 30%), which can be seen in Table 5.2. This is due to the fact that a mean value must be calculated, which involves two additional floating-point multiplications per node.

Data	recursive	iterative (# threads)				
size	1 thread	1	2	4	8	16
64^3	0.076	0.019	0.009	0.004	0.002	0.002
128^3	0.614	0.173	0.085	0.049	0.026	0.021
256^3	4.991	1.539	0.782	0.446	0.288	0.238
512^3	39.003	12.535	6.239	3.701	2.512	1.980

Table 5.2: Updating the semi-transparent tree in the threaded version shows the same scaling as for the iso-surface tree. The higher number results from the fact that computation of the mean values add multiplication operations.

So far, this analysis showed that data sets up to 256^3 can be updated interactively on a setup with 8 cores. For updating large volumes, more nodes are needed. Due to the linear scalability of the update routine, there is no significant barrier to using more processors.

5.1.2.4 Synchronization Mechanisms

Since updating the tree is now sufficiently fast for rendering time-dependent volume data sets, the missing part is an effective synchronization mechanism between the update and render nodes of the shared-memory system. As depicted in Figure 5.3, update and render nodes do not communicate with each other; rather both are controlled by a display master.

For distributing the rays among a number of nodes, this system relies on OpenRT [Dietrich03]. The rendered image is subdivided into tiles and the display master does load balancing by distributing the tiles among the specified number of nodes. Distribution of tiles for rendering is left to the

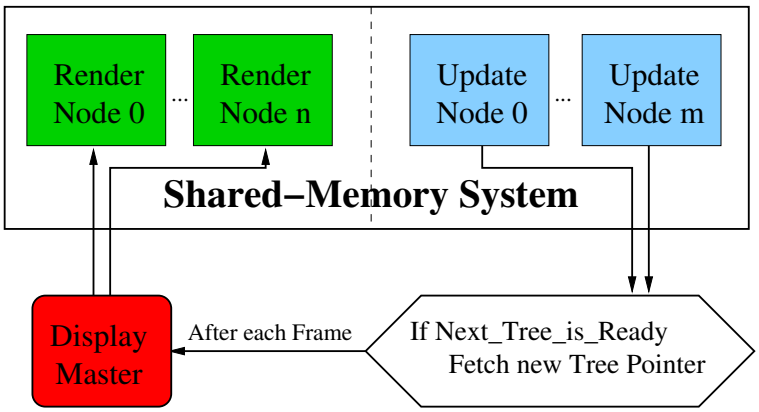


Figure 5.3: The shared-memory system is separated into render and update nodes, which do not communicate with each other since the display master controls the render and update tasks.

OpenRT system [Wald04a]. Unfortunately, OpenRT does not provide a dedicated display master, so this had to be added to the system. Here, on startup, semaphores are used to ensure that only one of the render nodes takes control of the update nodes.

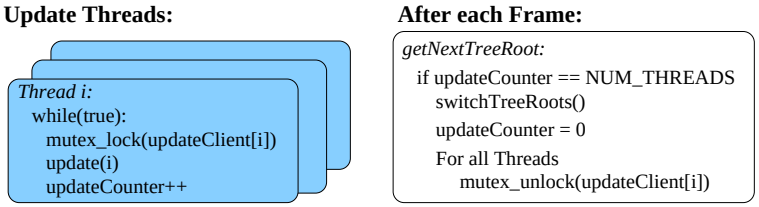


Figure 5.4: Each thread runs independently and increments a global counter, which is checked after each rendered frame. The display master unlocks the update threads for building the next kd-tree once the tree update is complete. Locking and unlocking of the counter is omitted.

Synchronizing the update threads is achieved by using mutex variables for locking, and a unique counter indicating how many update nodes have finished. Each thread locks its mutex before it starts updating. To enable rendering of time-varying data, two complete kd-trees are kept in memory for double buffering. One tree is rendered while the other tree is updated. When the display master asks whether a new tree is available to render, i.e. by querying this unique counter, the root addresses of both trees are

switched. Furthermore, all mutex variables are unlocked so that the next tree can be built. If the building process is still ongoing, the old tree pointer is returned, i.e. the current time step is rendered again. Otherwise the entire system would be blocked, making navigation tedious for the user in the case where the update is slower than the frame rate. Figure 5.4 sketches the implementation overview.

5.1.3 The 4D kd-Tree

A concurrent tree update may not in all cases be the best choice, since data sets larger than 512^3 requires probably 32 processors or more for an interactive update. On the other hand, the number of time steps might be only a few. It would than be better to incorporate the temporal domain directly into the kd-tree which is pursued in this section.

The idea presented here is similar to Woodring et al. [Woodring03], since space and time are also not treated as separate entities. To achieve this, the minimum and maximum values are not only computed for the spatial but also for the temporal domain. The temporal domain can be treated just as the spatial domains, i.e. always split the largest dimension in the middle. In the same way, the split alternate not only between the three spatial but also in the temporal domain.

5.1.3.1 Extending Tree Building and Traversal

Since the building routine is only used in a pre-processing step, it is left in its recursive implementation for simplicity. At the leaves of the tree, all values fetched from the volume data set need an additional index indicating the time step currently rendered. For inner nodes, the split still occurs in the largest dimension, regardless of whether it is spatial or temporal.

Similar modifications now must be applied to the traversal procedure (see Section 4.3.1). While the loop as well as the test (e.g. for minimum and maximum in the case of iso-surfaces) stays the same, the part for calculating the new split position must distinguish between spatial and temporal domains. Temporal splits must not partition the volume further, but instead partition the time step series.

The memory consumption is the same as storing for each individual time step a complete kd-tree and therefore rather disappointing. Even worse, the rendering performance decreases by 50 %. In other words, a rendering system which just loads a pre-computed kd-tree of the current time step into the system memory would be twice as fast but consume the same amount of memory. It is, however, believed, that the 4D kd-tree has a lot of room

for improvements, which will be demonstrated by implementing a simple but effective optimization in the next section.

5.1.3.2 Optimizing the Order of Splitting Planes

The reason for the poor performance of the naïve implementation is obvious. The additional temporal encoding destroys the efficient culling of subtrees if the distribution of the scalar values is similar over several time steps. This is almost always the case since the time between two steps is chosen to be sufficiently small to allow for a useful analysis of the simulation.

One possible optimization therefore seems to be to rearrange the splitting planes for the entire tree. To achieve this, the minimum and maximum values for all four possible split dimensions are computed, allowing the dimension with maximum overlap of both children's ranges and hence maximum culling efficiency to be selected. The rest remains as described in the original implementation, i.e. the orientation of the splitting plane is the same for each tree level and always splits in the middle (see Section 4.3.1).

Data size	Time Steps	Framework	fps (# threads) 1	2	4	8	12
Drop	100	static	3.36	6.80	13.40	27.02	38.99
512^2		4d-kd tree	2.58	5.25	10.17	21.15	30.13
5 Jets	200	static	1.70	3.61	7.36	14.71	21.53
512^2		4d-kd tree	1.68	3.78	6.92	9.82	14.24
Vortex	100	static	1.68	3.26	6.56	13.22	19.54
512^2		4d-kd tree	1.23	2.42	4.92	9.82	14.24
Drop	100	static	0.85	1.67	3.37	6.75	9.82
1024^2		4d-kd tree	0.56	1.36	2.47	5.31	7.79
5 Jets	200	static	0.43	0.85	1.66	3.85	5.60
1024^2		4d-kd tree	0.43	0.78	1.58	3.85	4.92
Vortex	100	static	0.43	0.85	1.66	3.35	4.95
1024^2		4d-kd tree	0.35	0.68	1.26	2.51	3.89

Table 5.3: Comparing the frame rates between the original static implementation and the 4D kd-tree shows a performance loss of approximately 30 - 40%. All numbers refer to the single ray implementation. This meets the expectation since the temporal kd-tree levels increase the size of kd-tree by the same amount.

To ensure that the optimal splitting dimension is chosen throughout the level, the arithmetic mean over all overlaps for each dimension is computed.

The splitting dimension, where the arithmetic mean is the highest is than chosen as optimal splitting dimension.

The overall system performance in frames per second (fps) for single-ray traversal was measured using the following data sets. The *Falling Drop* data set consists of 106x108x110 voxels and 100 time steps. As suggested in Wald et al. [Wald05], the volume is expanded only virtually to 128^3 to build a balanced tree. The second data set is a *Turbulent Vortex* consisting of 128^3 voxels and 100 time steps, while the third data set are *Five Jets* (128^3) with 200 time steps. Figure 5.5 shows all examples used.

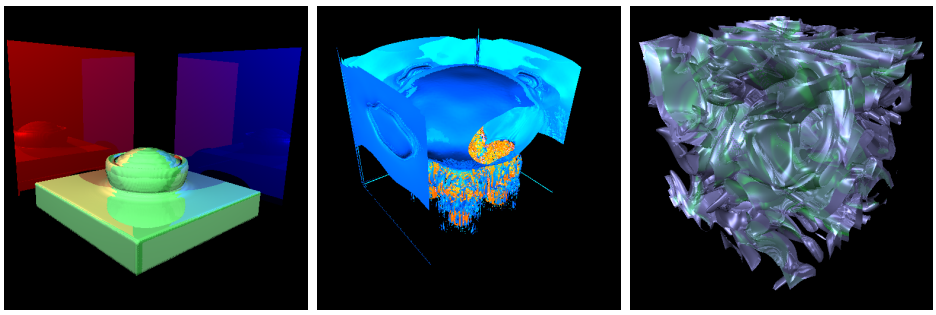


Figure 5.5: From left to right: The Falling Drop *in a polygonal environment with two light sources, shadows and reflections rendered at 7 fps (left).* 5 Jets *with two individually colored iso-surfaces and reflection rendered at 3 to 6 fps (middle). The* turbulent vortex *data set rendered as an iso-surface with reflection and translucence at 6 fps (right).*

The performance gain between this optimization and taking the dimension with the largest size as next split lies between 30 and 50% [Brauchle06], shifting the performance to 70 % of the original static implementation, as Table 5.3 shows. The missing 30 % result from the overhead introduced by traversing the temporal domain.

5.1.4 Comparison of Performance

To allow a direct comparison between the two approaches discussed in Section 5.1.2 and 5.1.3, a shared-memory with eight dual-core 2 GHz Opterons 870, each with 64 GB of main memory running Linux was used. All data sets stay the same as described in the previous section of this chapter.

The rest of the system stays as close as possible to the original implementation discussed in Chapter 4. This especially implies the use of the *small* variant of the kd-tree including all described optimizations (see Section 4.3.2)

and linear interpolation as cell iso-surface intersection (see Section 4.3.3). In every case a directional light shader was used for rendering the image. The screen resolution was set to a viewport size of 512^2 and 1024^2 for all measurements. Unfortunately, only a single-ray implementation was available for the 4D kd-tree. Therefore Table 5.4 presents the single-ray performance comparing all three system before Table 5.5 make use of SIMD but compares only the static with the concurrent approach.

		fps (#render nodes/#update nodes)					
Data Set	Framework	2/2	4/4	8/8	12/4	16/8	16/16
Drop	static	6.54	13.14	26.30	38.99	50.60	50.60
512^2	concurrent	6.30	12.39	21.73	30.34	25.14	24.30
	4D kd-tree	5.25	10.17	21.15	30.13	-	-
5 Jets	static	3.59	7.24	14.48	21.53	28.40	28.40
512^2	concurrent	3.66	6.98	13.16	18.43	16.46	16.10
	4D kd-tree	3.78	6.92	13.81	18.14	-	-
Vortex	static	2.77	5.60	11.10	19.54	22.0	22.0
512^2	concurrent	2.92	5.84	10.88	15.45	14.32	13.69
	4D kd-tree	2.42	4.92	9.82	14.24	-	-
Drop	static	1.65	3.12	6.65	9.82	12.80	12.80
1024^2	concurrent	1.57	3.176	5.75	8.94	8.92	8.71
	4D kd-tree	1.36	2.47	5.31	7.79	-	-
5 Jets	static	0.91	1.83	3.68	5.60	7.14	7.14
1024^2	concurrent	0.85	1.84	3.47	6.77	5.88	5.53
	4D kd-tree	0.78	1.58	3.85	4.92	-	-
Vortex	static	0.71	1.41	2.84	4.95	5.57	5.57
1024^2	concurrent	0.75	1.48	2.80	4.87	5.18	4.72
	4D kd-tree	0.68	1.26	2.51	3.89	-	-

Table 5.4: Performance comparison between the original static iso-surface renderer, concurrent build of kd-trees, and the extended 4D kd-tree using the single ray implementation. The header shows #render nodes / #update nodes (#update nodes are only applicable for the concurrent build). The performance is comparable between all approaches for up to eight clients. Update and render tasks compete on one node in the last two columns, which diminishes the performance.

The original system described by Wald et al. [Wald05] neither shares the kd-tree nor the data set between the nodes; so it was kept as is. The same is true for the 4D kd-tree implementation presented in Section 5.1.3, although it would be easy to adapt both systems to support shared-memory.

Additionally, the proposed optimization for the 4D kd-tree (see Section 5.1.3) was also not used here since this would lead to an incomparable comparison with the original implementation.

		fps (#render nodes/#update nodes)					
Data Set	Framework	2/2	4/4	8/8	12/4	16/8	16/16
Drop	static	13.25	26.95	52.33	77.84	81.20	81.20
512^2	concurrent	12.29	24.34	33.76	49.79	36.51	31.33
5 Jets	static	7.17	14.43	28.80	39.26	52.13	52.13
512^2	concurrent	7.28	14.61	22.73	33.97	26.38	22.08
Vortex	static	5.35	10.56	21.39	34.99	49.93	49.93
512^2	concurrent	5.47	10.87	17.46	27.46	21.31	19.11
Drop	static	3.80	7.42	14.60	21.70	26.23	26.23
1024^2	concurrent	3.59	7.08	11.80	17.23	14.88	14.47
5 Jets	static	2.12	4.26	8.55	12.12	16.66	16.66
1024^2	concurrent	2.22	4.13	7.71	13.01	10.32	10.24
Vortex	static	1.66	3.24	6.46	10.80	13.92	13.92
1024^2	concurrent	1.67	3.33	6.14	8.99	8.77	8.38

Table 5.5: Performance comparison between the original static iso-surface renderer, concurrent build of kd-trees, and the extended 4D kd-tree using the packet ray implementation. The header shows #render nodes / #update nodes (#update nodes are only applicable for the concurrent update). The results are similar to the single ray traversal, except that SIMD brings an overall performance speedup factor of approximately two to three.

The last implementation in this comparison relies on the concurrent tree update described in Section 5.1.2 and therefore, of course, shares both the data set and the kd-tree. Please keep in mind, that due to the double-buffering approach described in Section 5.1.2.4 two kd-trees are always kept in the system memory (representing the current and the next time step).

Table 5.4 directly compares the two newly developed approaches for time-dependent data rendering against the original implementation. From a theoretical point of view, concurrent building should result in about the same frame rate as the original implementation. This is true for up to eight clients. Using twelve render and four update nodes still improves the frame rate, but not as much as expected. The small number of update clients become the bottleneck here since the system was blocked for this measurement until the tree was updated to allow a useful comparison. It still turns out to be the fastest combination since, when using 16 render nodes, render and update tasks have to compete on one and the same node, which can actually reduce

the frame rate again. Scheduling becomes an issue, even if the update with 16 nodes is quite fast for such small individual time-steps (see Section 5.1.2.2). In essence, the performance scales linearly with the number of render clients as long as render and update clients do not compete with each other on a node. The 4D kd-tree is about 33 % slower compared to the static and the concurrent implementation due to the additional temporal nodes, which was already expected.

Table 5.5 shows the same performance measurements for the packet ray implementation. Everything else stays the same as previous described. Therefore it is no surprise that all interpretations for the single ray version are still applicable.

The performance gain matches those from the static iso-surface renderer, e.g. using SIMD increases the performance by a factor of approximately two for 512^2. A factor of four cannot be expected since the size of one individual time step is rather small, and SIMD works best with large screen resolutions and zooms. Therefore the speedup is significantly better when using a 1024^2 viewport resolution. Figure 5.6 on the following page shows individual time steps for each of the tested data sets. These images were rendered with an enhanced system allowing Phong shading [Phong75], including reflection and translucence, and supporting several light sources and cut views.

5.1.5 Results

As already mentioned, a shared-memory system was used for time-dependent volume rendering. The system consisted of eight dual-core Opteron 870 processors with 64 GB of main memory running Linux. Similar to the performance comparison between the concurrent tree update and 4D kd-tree in Section 5.1.4, Table 5.6 compares the single and packed ray performance between the static renderer and the concurrent tree update. Overall performance measurements in frames per second (fps) reveal that performance loss with respect to the original single-ray SIMD implementation is negligible as a long as sufficient number of update clients is available.

Configurations included two render / two update nodes and twelve render and four update nodes. This latter configuration delivers better performance since it uses a suitable relation between render and update nodes among the available processors. The rendering frame rates shows a similar behavior as the original system (see Section 4.3.1.3), i.e. the system scales linear with the number of processors. However, the speedup of packet traversal does not show as much variation as the original implementation does. In fact, the SIMD speedup is with a factor of approximately two rather stable. This is probably due to the fact that all data sets are similar with respect to

Data Set	Framework	2 rnd/2 upd nodes			12 rnd/4 upd nodes		
		Single	SIMD	Ratio	Single	SIMD	Ratio
Drop	static	6.54	13.25	2.0	38.99	77.84	2.0
512^2	concurrent	6.30	12.29	2.0	30.34	49.79	1.6
5 Jets	static	3.59	7.17	2.0	21.53	39.26	1.8
512^2	concurrent	3.66	7.28	2.0	18.43	33.97	1.8
Vortex	static	2.77	5.35	1.9	19.54	34.99	1.8
512^2	concurrent	2.92	5.47	1.9	15.45	27.46	1.8
Drop	static	1.65	3.80	2.3	9.82	21.70	2.2
1024^2	concurrent	1.57	3.59	2.3	8.94	17.23	1.9
5 Jets	static	0.91	2.12	2.3	5.60	12.12	2.2
1024^2	concurrent	0.85	2.22	2.6	6.77	13.01	1.9
Vortex	static	0.71	1.66	2.3	4.95	10.80	2.2
1024^2	concurrent	0.75	1.67	2.2	4.87	8.99	1.8

Table 5.6: Overall performance data (in fps) for the time-dependent renderer on a shared-memory system. The first number represents the number of render nodes, while the latter is the number of update nodes. SIMD traversal doubles the performance in all cases. Furthermore, large screen resolution from the same viewpoint increases the size of rendered cells and therefore generally results in a higher payoff.

their sizes. Comparing different screen resolutions further shows that packet traversal works in favor of larger viewports, which was observed in the original implementation. Note that the same framework can be used in conjunction with semi-transparent rendering (see Section 4.5). Since the performance mainly depends on the chosen threshold, and hence, on the quality desired, no performance measurements are presented here.

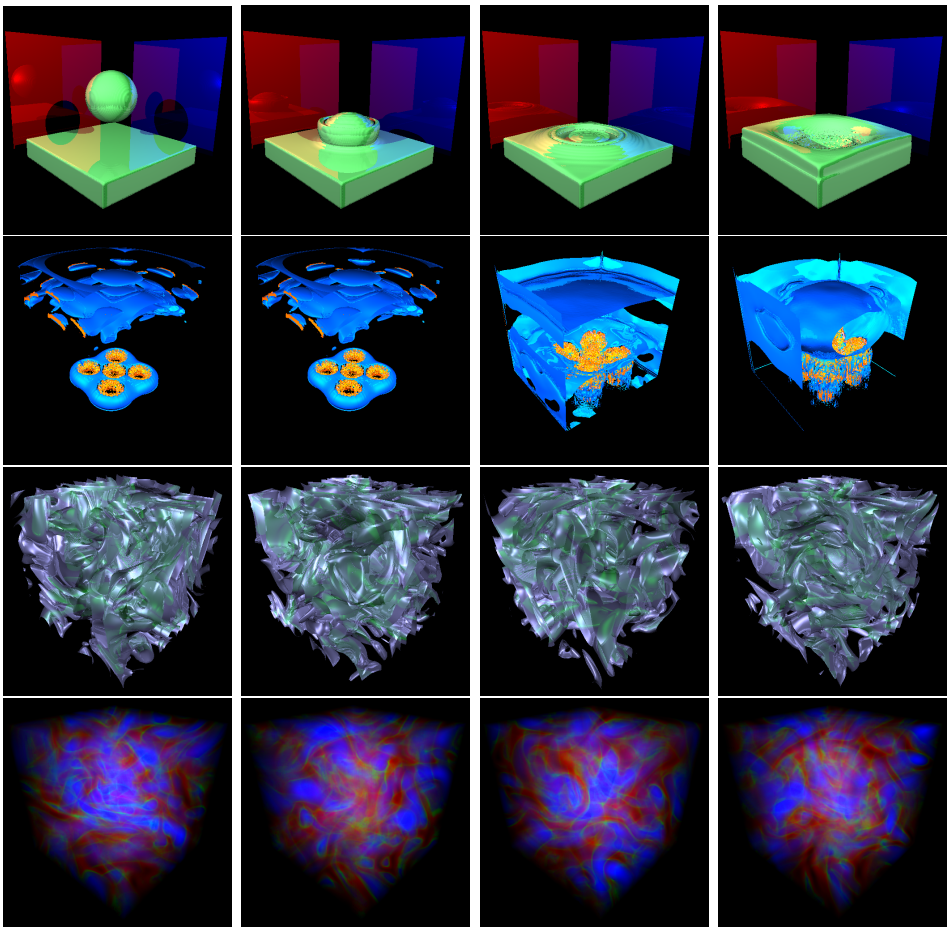


Figure 5.6: (a) A series of different time steps visualizing the Falling Drop *in a polygonal environment. The shadows are cast from two light sources. (b) The* Five Jets *with two individually colored iso-surfaces and reflection. A freely orientable cutting plane reveals the interior during rendering. (c) The* Turbulent Vortex *with two individually colored iso-surfaces. Reflection and translucence are used to make the second surface (green) visible. (d) The* Turbulent Vortex *data set rendered semi-transparent with a transfer function to highlight regions of interest.*

5.2 Other Applications I: Terrain Rendering

So far the implicit kd-tree has been used for rendering volumetric grid data only. By adding minimum and maximum values denoting the range of a subvolume in each node, the kd-tree is able to find cells containing an implicit surface based upon a user-defined value, the so-called iso-value. The kd-tree was chosen to allow for an efficient SIMD implementation. This was possible since the kd-tree is based on binary decisions. Massive volumetric data sets are supported using *treelets*. Subsequent sections showed that this acceleration structure can also be used for semi-transparent rendering by storing the average density and variation within the nodes. Furthermore, the kd-tree can be expanded to four dimensions for handling time varying data.

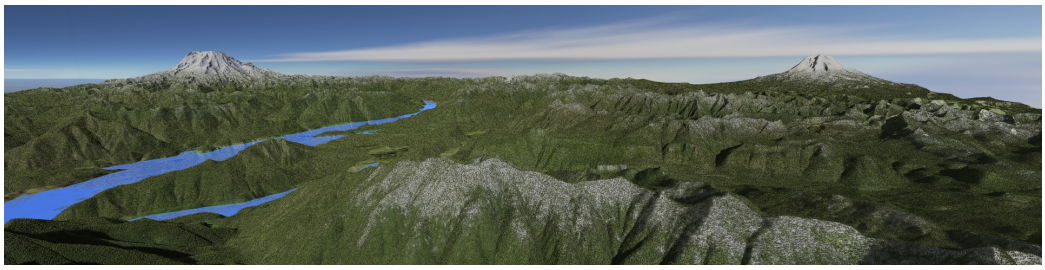


Figure 5.7: A panoramic view over a highly complex model of the Puget Sound Area. The ground terrain consists of 134 million triangles. It is covered with billions of plant instances, where each plant model is made up of several thousand polygons.

However, the *implicit kd-tree* has only been used for volumetric data so far. This section will discuss an alternative application for *implicit kd-trees* in the area of terrain rendering. In the following, a multi-level instancing approach is described, where the implicit kd-tree plays an important role. The final goal of this joint work with Andreas Dietrich [Dietrich06] was to render extremely huge landscapes covered with trees and forests (see Figure 5.7), where a user can freely choose between highly detailed close-up views or flyover scenarios.

The ground terrain alone consisted of 134 million triangles, while the billions of plant instances consisted of over 90 trillion triangles. Such high detail can hardly be achieved without instancing. Instancing basically means that only a pointer to a detailed model is stored so that the model can be used several times in the scene with negligible memory consumption. Whenever

a ray traverses a scene, therefore, it is simply redirected such that an intersection test with the detailed model can be applied. This technique is used here on two levels.

5.2.1 Wang-Tiling Scheme

In the first level, a so-called rectilinear tile of 160 m^2 is populated with trees, flowers, grass, and ground vegetation. Note that even for such a small tile, instancing is hardly avoidable. Several hundred trees, consisting of to 100,000 triangles each, can be found within the tile. Flowers or ground vegetation are placed several thousand times onto the tile to produce some sort of realistic landscape. Hence both types profit from instancing. Using affine transformations like rotation and scaling, the user has the impression that not every tree is identical to all the others. Of course, a number of different trees, bushes, flowers, and other plants was used to improve realism.

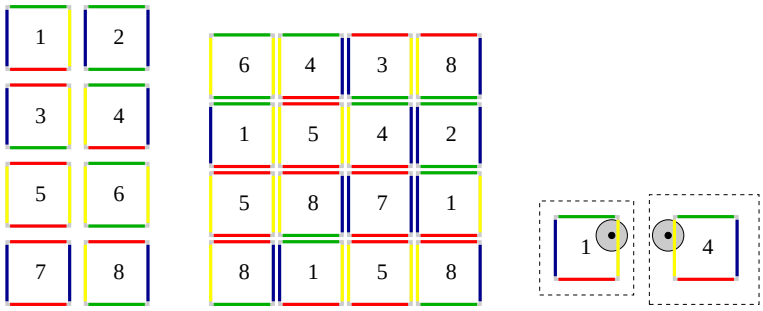


Figure 5.8: This Wang tiling scheme is used to cover the ground terrain with sub-scene plant tiles. Left: *Example with two different horizontal and vertical colors.* Middle: *An aperiodic tiling example.* Right: *Replication mechanism for overlapping plants.*

This rectilinear tile could now be distributed over a larger terrain by simple replication (i.e. instantiation). However, this would lead to a clearly visible pattern at the border of each tile. Instead, it is better to use a *Wang* tiling scheme, illustrated in Figure 5.8. In this example, 18 tiles are generated and each border is assigned a certain color. When distributing the tiles on the ground terrain, all tiles with adjacent borders must have the same color to achieve an aperiodic tiling scheme. In other words, all tiles must be designed in such a way that plants may overlap whenever they have the same color at an adjacent border. More details about aperiodic tiling can be found in [Wang61, Cohen03, Deussen05]. The focus will now turn to the ground terrain where the tiles are actually placed.

5.2.2 Ground Terrain Traversal

This ground terrain is basically an elevation map where an elevation z is assigned to each position in x and y. All plants on this elevation map must be placed with respect to the actual elevation on the map. It seems plausible to partition this elevation map into cells that correspond to the tile size so that each cell can be associated with a previously described tile. This leads to the issue of quickly finding the terrain cells intersected by a ray. These cells can be seen as a 2D grid containing minimum and maximum z (i.e. elevation) values for each cell bounding box.

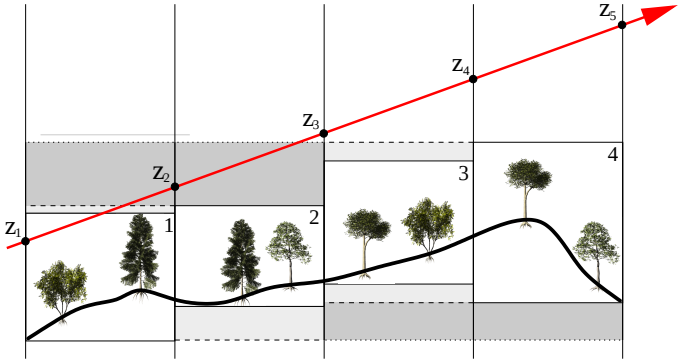


Figure 5.9: Top-level terrain kd-tree traversal. Dashed and dotted lines indicate how min/max z-values of kd-tree leave nodes can be combined recursively to form the min/max z-values for inner nodes.

Hence, a 2D grid traverser could be employed by performing some sort of 2D line drawing algorithm. While marching through the cells, the minimum and maximum values attached to each cell would be compared against the z-value of the current ray, evaluated at the entry and exit points. The cell can be skipped if both the entry and the exit values lie above the cell's maximum value or below the cell minimum value. See Figure 5.9 for an illustration.

This problem appears quite similar to the iso-surface rendering problem. Indeed, the kd-tree traversal described in Section 4.3.1 needs only a small modification at a certain place, i.e. by exchanging the line:

$$\rho_{iso} \geq node(\rho_{min}) \wedge \rho_{iso} \leq node(\rho_{max})$$

with

$$D_{near_z} \geq node(\rho_{min}) \wedge D_{near_z} \leq node(\rho_{max}) \vee$$
$$D_{far_z} \geq node(\rho_{min}) \wedge D_{far_z} \leq node(\rho_{max}).$$

The kd-tree traversal is now ready for terrain rendering. The building routine is even easier, since the split occurs only in two dimensions, x and y. By using a kd-tree, larger parts of the terrain can be hierarchically skipped. The minimum and maximum values of the leaves contain the minimum and maximum elevation values for z within that cell. As for the *implicit kd-tree*, inner nodes contain 'merged' values of its left and right child, i.e. $min_{father} = min\{min_{child_l}, min_{child_r}\}$ and $max_{father} = max\{max_{child_l}, max_{child_r}\}$. Hence, a child can be skipped if a ray lies above the top (or below the bottom) bounding surface of an inner node. For example, in Figure 5.9, z_3 to z_5 are above the maximum z-value of cells three and four, and therefore, these cells do not have to be investigated further.

Similar to the implicit kd-tree, the split is always set to the middle of the largest dimension. More elaborate algorithms, such as surface area heuristics (SAH), could determine a better splitting plane position by using cost prediction functions.

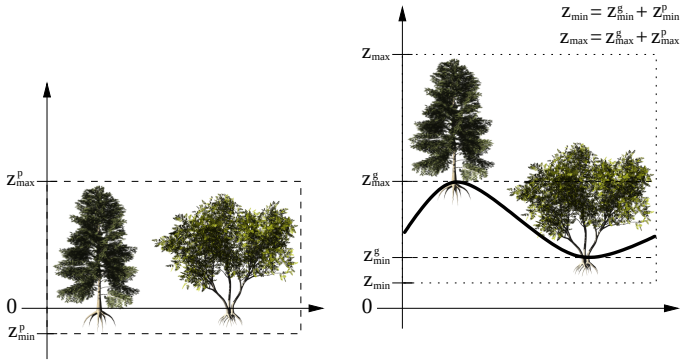


Figure 5.10: Approximate calculation of the min/max z-values of terrain bounding boxes. Left: *Bounding box of a plant sub-scene.* Right: *The ground bounding box extended by the sub-scene bounding box.*

If the cell is populated with plants, it is not possible to simply take the minimum and maximum values of the elevation of the cell. The plants of the sub-scene placed into the elevation cell must also be taken into account, as illustrated in Figure 5.9. Suppose, all plants are aligned to the ground, i.e. all plant roots have negative coordinates, then the minimum and maximum z-values of the plant's bounding box (z^p_{min} and z^p_{max}) are simply added to the minimum and maximum values obtained from the elevation positions (z^p_{min} and z^p_{max}), resulting in

$$z_{min} = z^g_{min} + z^p_{min}, \qquad z_{max} = z^g_{max} + z^p_{max}. \tag{5.1}$$

This estimation led to over-conservative cell bounding boxes, which can be avoided by calculating z_{min} and z_{max} based on the actual plant z-position in the respective cell. Note, however, that a correct calculation involves the testing of potentially billions of plant instances. Also, handling negative coordinates can be neglected if the camera will not move below the ground. Once it is determined that the ray intersects a cell, plants and ground terrain are then tested against the ray separately. It turns out that it is better with respect to performance to first test against the ground scene to obtain useful near and far values for the current ray, and subsequently, test all plants. This avoids permanent switching between two traversal operations (terrain and plants) once a ray enters the cell.

A two-dimensional lazy kd-tree is used for traversing the plants within a cell. This is a top-level kd-tree, since each geometric object maintains its own 3D SAH kd-tree. Only the bounding boxes are organized in this second-level kd-tree, which allows instantiation at the same time. Care must be taken, however, that the plants are shifted in the z-dimension to place them properly onto the elevation map. Omitting horizontal splitting planes in this tree completely reduces the traversal to merely extending the upper bounding box z^p_{max} value by $z^g_{max} - z^g_{min}$. When entering a leaf of this kd-tree, the plant's z-direction is adapted to the local height map coordinates.

Of course, the entire implementation for rendering realistic looking landscapes is far more complex. For example, it certainly does not suffice to use an aperiodic tiling scheme since humans expect different plants on certain grounds and heights. An adaptive plant density reduction scheme avoids the most significant flaws, e.g. non-water plants in water, fewer plants where snow is found and a decreasing plant population depending on the height. For details refer to [Dietrich06].

5.2.3 Results

This highly detailed landscape suffers from rather poor rendering performance. Using a shared-memory system of eight dual-core Opterons with 64 GB RAM and video resolution, still the system needs several seconds to render an image. Table 5.7 shows the impact of different orders of magnitude of triangles on the resolution of the underlying terrain grid.[2] This 2D grid distributes the pre-computed Wang tiles over the landscape.

Each tile was designed so that the plants, fit best with a 512^2 resolution. Reducing the grid resolution therefore destroys the correct relation between

[2]The measurements in [Dietrich06] are already outdated, since resolved bugs and other enhancements improved the rendering speed tremendously.

landscapes and plants, i.e. trees, bushes, and flowers appear to large compared to the ground. This is of course not realistic anymore but irrelevant for the performance measurements. All frame rates were measured at video resolution (640×480). Due to the logarithmic scaling, changing the grid resolution has almost no impact on the rendering time.

Grid Resolution	Triangles	Frame time (sec)
64×64	$1.4 \cdot 10^{12}$	4.39
128×128	$5.7 \cdot 10^{12}$	4.44
256×256	$22.6 \cdot 10^{12}$	4.90
512×512	$90.5 \cdot 10^{12}$	5.18

Table 5.7: Rendering performance for different terrain grid resolutions and scene complexities. The last line contains the correct terrain to plant relation.

The sub-scenes populating the scene consisted of approximately 82,000 plant instances (the exact number may differ from tile to tile since a *Poisson-Disc* distribution was used). Each plant itself consisted of another 1,000 to 100,000 triangles, leading to 454,000 triangles per tile. Due to the extensive use of instancing, the data geometry and kd-tree consume only about 1 GB of main memory.

Nevertheless, this application shows, that the implicit kd-tree is a versatile acceleration structure. Originally developed for iso-surface rendering on regular grids, it can now be used for semi-transparent rendering, rendering of time-dependent data, and was recently adapted to terrain rendering with only minor modifications.

5.3 Other Applications II: Dynamic Rendering

The last section discussed a slight modification to the *implicit kd-tree*. The range test found in the innermost loop was adapted so that it quickly determines the terrain cells pierced by a ray. In this section, the *implicit kd-tree* is going to be modified to allow an efficient traversal of triangles in dynamic scenes [Woop06].

Recall that the *implicit kd-tree* stores the minimum and maximum value, i.e. the range of the iso-values of the associated sub-volume in each node. However, this tree traverses triangles and not scalar values. The minimum and maximum values are therefore replaced by a plane where the plane orientation alternates between the x, y, and z dimension. Both plane positions are determined by the triangles bounding box at the leaf level for a certain

orientation. Higher tree levels not only alternate the plane orientation but merge the triangles bounding boxes up to the scene bounding box stored in root node of the kd-tree.

In an alternative interpretation, the *bounded kd-tree* is a hybrid spatial index structure. It combines the advantages of bounding volume hierarchies with those of the kd-trees in a single homogeneous data structure. Bounding volume hierarchies efficiently support dynamic scenes, but are more costly to traverse, especially on hardware. Ordinary kd-trees on the other hand can be efficiently traversed even with packets of rays but lack support for dynamic updates. Bounded kd-trees combines the strengths of both acceleration structures into a single one and can be defined as follows [Woop06]:

> A *Bounded kd-tree* is a binary tree, where each node recursively subdivides the geometry of the scene into two disjoint subsets represented by its two children. Each node stores the index of a coordinate axis and bounds on the geometric extent of its two children along this axis in the form of two intervals, one for each child, often also referred to as a slab (see Figure 5.11). Each leaf node stores a reference to a single primitive of the scene.

In other words, a scene with N primitives has exactly N leaf nodes and $N-1$ inner nodes. Hence, neither a special handling of primitive lists is necessary nor is the size of the tree unpredictable. The bounding planes still can be efficiently updated but the size of the data structure is reduced by a factor of three compared to bounding volume hierarchies since only one dimension is stored per kd-tree level.

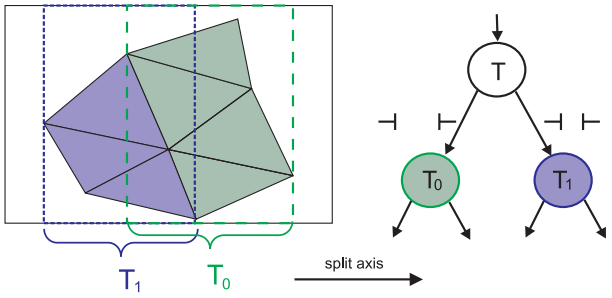


Figure 5.11: Each inner node is split into two overlapping subsets (T_0 and T_1 in this example). Note that only one splitting axis is stored per level. This subdivision continuous until there is only one primitive per node.

From the construction follows furthermore that the traversal of *bounded kd-trees* is strictly ordered along the ray, i.e. if an intersection occurs in front

of the next node, the traversal computation can be terminated early. The traversal order of the child nodes can even be precomputed for the axis and depends solely on the sign of the ray direction.

5.3.1 Update and Traversal Process

Since the geometry is allowed to change over time, an efficient updating mechanism is required. The update mechanism is in its core similar to the concurrent *implicit kd-tree* update described in Section 5.1.2. Instead of minimum and maximum values, this update process merges two different bounds of the nodes from bottom-up through the tree and updates for each *bounded kd-tree* node the extend of both children along the nodes' axis. Care has to be taken that the geometry in a sub-tree stays as close together as possible since a mismatch may lead to significant overlap of the bounds of child nodes and hence redundant traversal. As a result, only dynamic scenes with at least some coherence profit from this acceleration structure while the benefit for, e.g. for explosions, will be rather marginal.

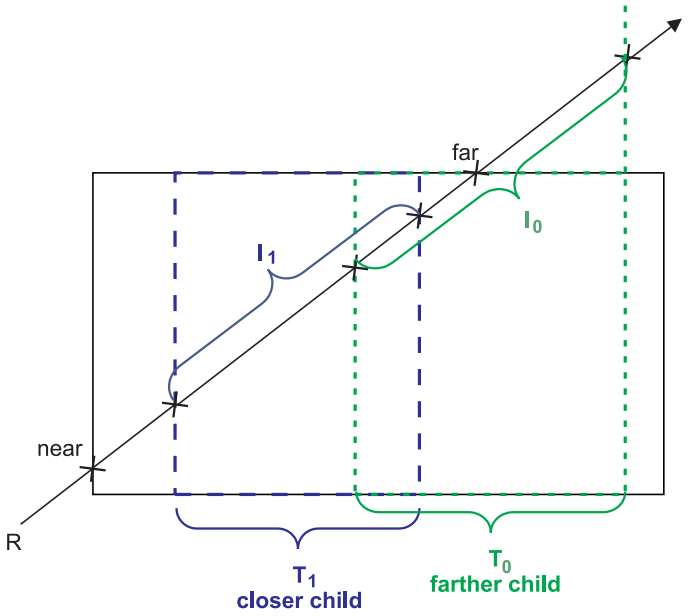


Figure 5.12: During each traversal step, the ray intersects four planes which are defined by the children bounds. This results in two (possibly overlapping) intersection intervals $I_{\{0,1\}}$ along the ray. A child is only traversed iff its intersection interval overlaps the traversal interval defined by $I = [near, far]$. Examination of the closer child allows for early ray termination.

It should also be clear that traversal is very similar to standard kd-trees. The ray is first tested for early ray termination by comparing the hit distance with the *near* distance. Subsequently the ray is intersected with four bounding planes defined by the node, giving the two intersection intervals $I_{0,1}$ one for each child node. An example traversal is illustrated in Figure 5.12. Two comparisons first determine (for instance child 0) whether its intersection Interval I_0 overlaps the current traversal interval I. This child is recursively traversed with the interval updated to the intersection of I and I_0. In case that the other child overlaps the traversal interval is pushed onto the stack together with its Intersection of I and I_0 as its traversal interval.

In contrast to the algorithm described in Section 5.1.2.1, a top-down approach builds the first kd-tree including a Surface Area Heuristic to select an optimal partitioning of the triangle set. For details it is referred to [Havran01, MacDonald89, Woop06].

5.3.2 Results

The entire rendering is realized in hardware using an FPGA chip for rapid prototyping, .e.g. the Update and Traversal Units. Skinning and Geometry Units are other important parts responsible for recomputing the position of all vertices in a mesh for every frame and for sequentially intersecting the rays of a packet with the triangle geometry respectively.

Scene	triangles	DRT	SaarCOR	RPU	OpenRT
Scene6	0.8 k	45.0	44.6	20.8	12.9
Office	34.3 k	27.9	35.9	14.6	10.4
Gael	52.5 k	14.4	18.6	7.5	8.0

Table 5.8: A comparison with the original implementation shows a slightly slower performance caused by the more expensive intersection test. However, the dynamic approach is still 2 to 3 times faster compared to a 2.66 GHz Pentium-IV. The last four columns represent frames per second.

The prototype platform at that time was a Xilinx Virtex-II 6000-4 FPGA hosting an Alpha Data ADM-XRC-II PCI-board. The FPGA has access to an 64-bit wide DDR memory interface that can deliver a peak bandwidth of 1.0 GB/s at 66 MHz. More detailed on all implemented units and the architecture setup can be found in [Woop06]. Some sample images from rendered dynamic scenes are shown in Figure 5.13. All test scenes achieve interactive performance on our FPGA prototype.

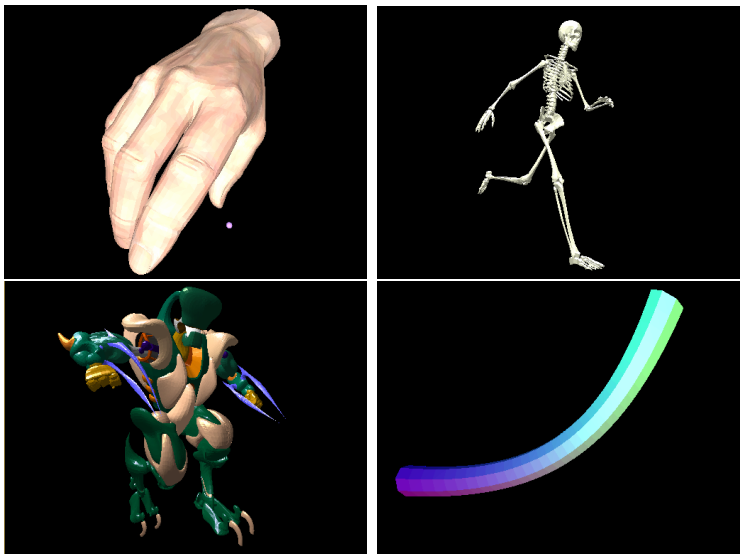


Figure 5.13: Sample images of dynamic scenes rendered in real time with 10 to 56 fps at 512 × 384 *screen resolution on both prototype FPGAs.* Upper-left: *Hand (17k triangles, 33 fps),* upper-right: *Skeleton (16k triangles, 34 fps),* lower-left: *Helix (78k triangles, 10.5 fps), and* lower-right: *Pipe (0.5k triangles, 56 fps).*

Restricting to static scenes, it is possible to compare this new prototype against the fixed-function SaarCOR ray tracing prototype, the programmable RPU architecture and the OpenRT ray tracing system. Table 5.8 shows that the rendering performance is slightly slower compared to the original fixed-function SaarCOR architecture. This is caused by the lower intersection performance (2.0 compared to 1.25 cycles per intersection). On the other hand, this dynamic ray tracer is still 2 to 3 times faster than the used 2.66 GHz Pentium-IV.

Dynamic scenes must be further distinguished in scenes with available skinning models or not. This explains the rather large variation for the *Hand*, *Skeleton*, and *Helix* scene with respect to the relative cycles of the Update Unit (See Table 5.9). All three are poser animations, thus the vertex positions must be precomputed by Poser and uploaded to the FPGA via DMA. Note that update is still performed in hardware. Other test scenes use typical deformation, e.g. bending and morphing. The *DynGael* scene contains the static gael environment with 25K triangles and several dynamic objects with 44k triangles. The *bounded kd-tree* nodes on top of these six object instances are recomputed by the driver for each frame.

Table 5.9 reveals relative cycles and frame rate for these dynamic scenes. All numbers refer to a 512×386 resolution for each of the demo scenes. Obviously, rendering is the most expensive operation since it occupies mor than 90 % of all clock cycles. It can further be inferred, that the Skinning Unit behaves linear in the number of vertices times the number of connected matrices. The random scenes shows the worst performance of all demo scenes since the random triangle distribution cannot be matched to the initial *bounded kd-tree* structure during the morphing sequence.

	skinning			relative cycles			fps
Scene	triangles	vert.	mat.	skinning	update	render	
Pipe	0.5k	0.26k	2	0.17%	0.21%	99.6%	32.6
Hand	17k	9.3k	-	-	6.8%	93.2%	20.9
Skeleton	16k	8.3k	-	-	7.0%	93.0%	25.5
Helix	78k	50.1k	-	-	13.7%	86.3%	12.3
Rot. Cube	18k	17.7k	2	4.5%	5.2%	90.2%	18.5
Morph	4.3k	2.1k	3	1.6%	1.8%	96.6%	32.3
DynGael	97k	184k	-	1.6%	3.6%	94.8%	13.0
Random	4.3k	12.9k	3	0.5%	0.2%	99.3%	2.8

Table 5.9: This table presents scene details including the relative cycle usage of each unit and the achieved frame rates. All measurements refer to a resolution of 512×386*. Interactive frame rates are in all but the random scene possible.*

Not demonstrated here, but in [Woop05] several FPGAs boards might be employed with linear scaling capabilities as long as each FPGA has a local scene description for traversal. Hence, dynamic scenes have to rely on a fast synchronization scheme if this feature is added. This data structure is robust, fast, and easy to use as long as the triangles are coherent. This should, except for explosions, rarely be the case and therefore be widely used.

5.4 Conclusions

This chapter covered three extensions to the original iso-surface renderer based on the *implicit kd-tree*. Time-dependent volume rendering was first enabled by a concurrent tree update. It was shown that replacing the recursive tree built by an iterative scheme, together with multi-threading enables the use of a number of nodes within a shared-memory system to update the kd-tree concurrently. Using a double buffering approach, one tree is traversed

for rendering while the other is updated. The performance can keep up with the original static implementation as long as additional nodes for updating are available. Although this approach itself limits rendering to rather small data sets, it certainly has a practical usage.

Since CFD simulations are even on todays super computers time consuming, it is often required to produce testing series with small resolutions. Such series could be instantly checked with the rendering system extended with the concurrent tree update presented here. Another idea is to use this approach for streaming applications, since the acceleration structure is independent of the number of time steps.

However, the data set size is limited this approach, which is why the 4D kd-tree was introduced. Currently it both consumes too much memory and is too slow due to the additional temporal encoding. Optimization approaches was, however, are promising, i.e. by using a more sophisticated splitting orientation scheme for the tree.

The next two sections showed that *implicit kd-trees* are not necessarily restricted to the area of volume rendering. They can also be used in the multi-level instancing approach, where the *implicit kd-tree* stores the minimum and maximum elevation heights instead of scalar iso-values. Culling parts of the terrain, which are not intersected by the ray improves speed and enables render times of less than five seconds for a video resolution image using a shared-memory system equipped with eight dual-core Opterons. It should be noted that the complete landscape consists of trillions of potentially visible triangles, which can hardly be handled by modern graphic cards. Additionally, advanced lighting effects can also be instantly used for this terrain renderer.

The *bounded kd-tree* for dynamic rendering of polygonal scenes can furthermore be seen as a variation of the *implicit kd-tree* as demonstrated in section 5.3. Bounded kd-trees proved as a good trade-off between update and rendering speed. This enable interactive frame rates even on an FPGA prototype as long as no complete random movement happens.

Overall, the kd-tree has proven to be a versatile tool, if the underlying data is organized in a 3D or 4D grid. Unfortunately, other types of volumetric data do not obey this strict rule (see Section 2.2.3). Regarding unstructured and semi-structured data, a new algorithm is needed that provides interactive performance. This will be covered in the next chapter, where the *Plücker space* is introduced allowing for an efficient cell-by-cell (i.e. incremental) traversal of tetrahedral and hexahedral meshes.

5.5 Contributions

The author's contributions to the topics discussed in this chapter are:

1. The author completely developed and implemented the concurrent tree update mechanism in [Marmitt06a], including a fast and iterative tree update, a synchronization mechanism between render and update clients, and its integration into the OpenRT rendering framework.

2. The author developed the basic concept of the 4D kd-tree and supervised its implementation in a master thesis [Brauchle06].

3. The author adapted the *implicit kd-tree* for the terrain rendering system [Dietrich06], implemented the Wang-tiling scheme, and generated the landscapes for performance measurements and the award-winning image for the *Computer Graphics Forum Cover competition*.

4. The author implemented several important parts of the FPGA-based polygonal rendering system [Woop06] including a compiler for the kd-tree update mechanism and a fixed function shader. He was furthermore responsible for creating the test scenarios and measurements.

5.6 Future Work

Although the concurrent tree update was optimized in many ways, its main drawback is its the restriction to small volume data sets. As already shown, fetching the data for the leaf level is still a bottleneck and prevents its usage for volumes larger than 512^3. One interesting idea is to build the kd-tree not for one but say for two or four time steps per update. Relaxing the constraint of having a kd-tree ready for every time step would give the update routine more time. The 4D kd-tree could for example used for this, assuming that it is developed with efficient encoding in mind.

Additionally, the test system had 64 GB of main memory so that all time steps were probably kept in main memory, even with enabled memory mapping. Linux memory mapping will, however, not suffice if the data set either consists of thousands of time steps or the rendering system is equipped with less memory. This could be true for upcoming low-cost systems with quad-core CPUs, which may be equipped with only 2 or 4 GB of main memory. Hence, a more efficient memory scheme handling seems unavoidable, as with the MMU introduced for rendering massive volume data sets [Friedrich07].

The 4D kd-tree was the second method proposed in this chapter for rendering time-dependent data sets. First results are unsatisfactory in terms

of rendering speed and memory consumption. Hence, this structure requires further investigations to become a competitive acceleration solution for time-dependent volume rendering. Its greatest advantage is that there is no time limit for constructing the tree, since the entire tree is built during pre-processing. In this way, the two proposed methods must be treated as complementary rather than competitive methods.

Finally, the use of the *implicit kd-tree* showed its versatility in the area of terrain rendering, as seen in Section 5.2. A variation, the *bounded kd-tree* can be used to efficiently render dynamic polygonal scenes (see Section 5.3). Therefore this chapter closes with an image shown in Figure 5.14 winning the *Computer Graphics Journal Image Competition 2007*. It was generated with the system proposed in [Dietrich06].

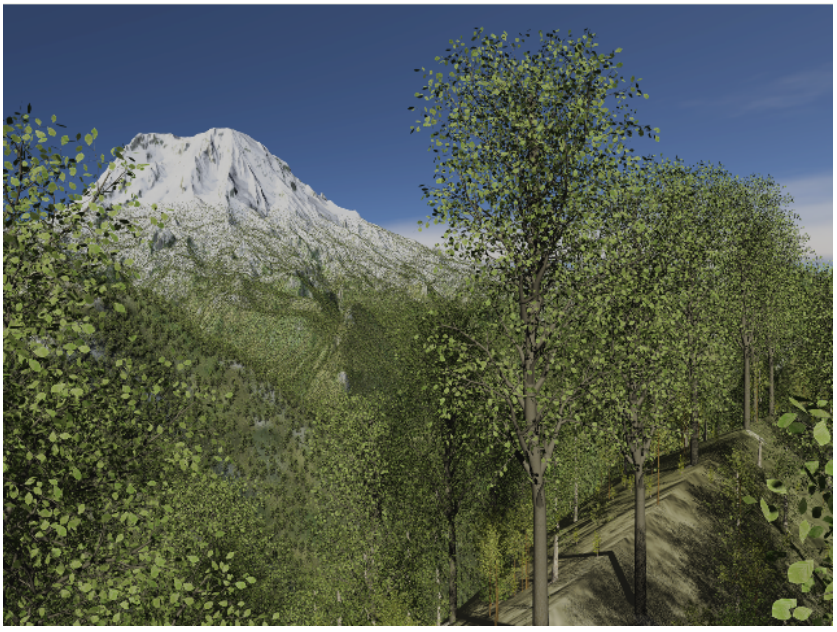


Figure 5.14: Example close-up view on some on the trees. All leaves are modeled as alpha-textured polygon meshes, which result in a high number of transparency rays. The scene is fully ray traced each frame, without any kind of precomputation or geometric simplification.

Chapter 6
Irregular Data Sets

If you have an apple and I have an apple and we exchange these apples then you and I will still have one apple. But if you have an idea and I have an idea and we exchange these ideas, then each of us will have two ideas.

George Bernard Shaw

Previous chapters extensively discussed the *implicit kd-tree* to accelerate iso-surface rendering of structured volumes. From this starting point, both the hierarchical structure and the traversal procedure were augmented to support massive data sets, as well as semi-transparent and time-dependent rendering of volumetric data. However, the restriction to regular grids was never lifted even though scientific simulations often rely heavily on semi-structured or even unstructured volumetric data.

These two types of volumetric organizations (see Section 2.2.3) are the focus of this chapter. Their irregular organization makes the visualization task far more demanding compared to data sets organized in regular grids. In contrast to previous chapters, no hierarchical approach will be presented, but rather a fast incremental traverser. Hierarchies are not as easy to implement as for regular grids due to the irregular organization of the underlying data.

An incremental traversal on the other hand offers full flexibility with respect to the visualization task (see Section 2.2.5), i.e. iso-surface rendering, maximum-intensity projection, and semi-transparent rendering. The main part of this chapter will therefore discuss a fast approach for traversing irregular volume data. Using the *Plücker test* for deciding, whether a ray passes clockwise or counter clockwise with respect to an oriented edge connected to values of the scalar field, a primitive-by-primitive traversal is easy to implement. Implementation and performance measurements are explained for tetrahedral primitives in Section 6.3.2. It should be noted, however, that *Plücker tests* can be applied to any convex polyhedra, e.g. a prism, making it a versatile and powerful method for incremental traversal of irregular data sets. A hexahedral traversal is explained in Section 6.4.2.1.[1]

6.1 Related Work

Garrity [Garrity90] and Wilhelms et al. [Wilhelms90] were the first to consider software ray casting for irregular grids. Garrity adapted a method introduced by Siddon [Siddon85], who describe an approach for ray tracing regular volumes using ray-plane intersections for determining all intersected

[1]Note that a hexahedron is not necessarily a convex polyhedra. In practice, however, many curvilinear grids consist of hexahedra with planar faces.

cells along a ray path. While traversing the volume, each cell is decomposed into six faces, where each face is treated as a plane. Given an entry face, i.e. the face where a ray enters a cell, the exit face can be determined by computing the ray-plane intersection for all five possible exit faces within a cell and choosing the face with the minimum distance. Since each face is shared by two cells, except for boundary cells, all subsequent cells along the ray can be determined. In a regular grid, the planes within one dimension are parallel and uniformly spaced which keeps the computational burden light.

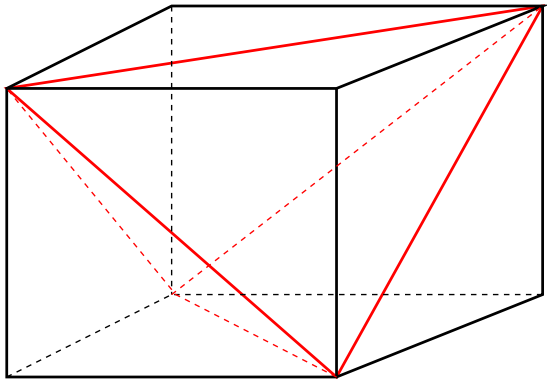


Figure 6.1: A convex hexahedron can be approximated by five tetrahedra if the renderer supports tetrahedral meshes only.

Garrity [Garrity90] propose in the same way an incremental traverser for tetrahedral meshes. Again, given the entry face of an intersected tetrahedron, the ray is intersected with all three possible exit faces where the intersection with the smallest distance to the current intersection is the sought exit face. Since tetrahedral faces are planar by definition, ray-plane intersection can be used here also. Except for the boundary faces, all faces are shared by exactly two tetrahedra. Additional overhead results from keeping track of adjacency information; therefore an index is attached to each tetrahedral face for all tetrahedra pointing to the tetrahedron sharing the face.

Sliding interfaces, i.e. faces that share only sub-areas of two adjacent primitives, are not allowed but hardly needed for tetrahedral meshes anyway. To render curvilinear grids consisting of hexahedral primitives, Garrity suggested to approximate each (convex) hexahedron by five tetrahedra. A sample decomposition for a convex hexahedra is illustrated in Figure 6.1.

The missing link to a complete volume renderer is how to find the initial tetrahedron along a ray. A naïve approach would be to intersect the ray with *all* faces of the volume and choose the minimum distance. A better

way is to use an acceleration structure. Garrity decided to use a grid in which all boundary faces were sorted. For the tetrahedral case, this is a set of triangles, where a pointer to the corresponding tetrahedra is attached to each boundary triangle. This two-step approach is typical and used often in such techniques, including the approaches discussed here (see Section 6.3 and 6.4). Garrity measured the performance on a Stardent ST1000 with four processors. Rendering a 512^2 viewport required 80 to 150 seconds for data sets consisting of up to 16,000 tetrahedra.

Wihelms [Wilhelms90] compares two basic principles when rendering curvilinear grids. The first approach renders the volume directly by decomposing all six faces of a hexahedron into two triangles per face, resulting in a total of twelve triangles per hexahedra. Assuming a convex hexahedra, at most two triangles are intersected per hexahedron, determining the entry and exit face. From that point, the inherent adjacency information can be used to incrementally traverse the curvilinear data. To interpolate the scalar value at the entry and exit faces, barycentric coordinates are used directly resulting in discontinuities at the diagonal line introduced on each face.

The other methods summarized in [Wilhelms90] interpolate (i.e. resample) the curvilinear volume into a regular volume. This can be achieved by either using tri-linear (see Section 2.2.4) or an inverse distance-weighted interpolation. The latter approach uses a neighborhood of points for resampling the grid values, where the influence of each point decreases with increasing distance to the resampled point. Nearest neighbor interpolation, the last method compared, is rather simplistic. The point in space closest to the point to be resampled is chosen without interpolating the scalar value, typically resulting in blocky rendered images. At that time converting the curvilinear to a regular grid was the best choice since the rendering time of a curvilinear grid of 800 seconds (128^2 viewport) was an order of magnitude higher on an SGI Iris 4D50GT compared to rendering a regular grid.

An interesting alternative was proposed by Frühauf [Frühauf94] for traversing curvilinear grids directly. Instead of traversing rays through the curvilinear grid, the curvilinear grid is warped into a regular grid, simplifying the incremental traversal tremendously. This mapping from physical space to computational space[2] is computed using a Jacobian matrix. Frühauf suggested using central differences to approximate this Jacobian matrix, i.e.

[2] *Computational space* is an abstract representation of the logical organization of a curvilinear (or even rectilinear) grid. It can be seen as a mapping from a warped (i.e. curvilinear) grid to a regular grid with an orthographic coordinate system and unit length cells for each dimension. This term originates from numerical simulations, since all computations are performed in this space. All results are then mapped back to *physical space*, i.e. the non-regular representation [Frühauf94].

using adjacent vertices in computational and physical space. Each vertex or vector can then be transformed from computational to physical space, and vice versa using this Jacobian matrix.

An incremental grid traversal for regular grids is then employed to resample the scalar values. At each face intersection of a grid cell, the ray is bent according to the precomputed vectors given by the four face vertices using bilinear interpolation. To reduce computational costs, all ray bendings are precomputed for each node and updated only when changing the viewpoint. Hence, if the mapping parameters, e.g. transfer function, changes, the ray paths can be directly used without any further computation. To find the initial cell, all boundary faces are color-coded and projected onto the screen using graphics hardware [Weghorst84]. Unfortunately, Frühauf reported no performance results for his system.

Ma [Ma95] was one of the first to exploit the parallelism of ray tracing for rendering tetrahedral data. The data set, as well as the rendering, is distributed among all processing nodes. In a preprocessing step, the volume is partitioned such that each processor handles only a subvolume. Boundary faces are projected orthographically onto the screen to determine the first cell along a ray. Incremental traversal is identical to [Garrity90].

Transfer functions lead to artifacts if interpolating only at the tetrahedral faces. This is mainly caused by not respecting the Nyquist limit during sampling and can be avoided by calculating sufficient samples in-between the tetrahedra. Instead of an adaptive sampling scheme, Ma [Ma95] chose only one additional point per tetrahedron and applied a trapezoidal filter between each pair of sample points. All processors accumulate the interpolated values along a ray independently. These contributions must be sorted with respect to visibility order before the final image can be composed. Ma measured the performance of a synthetical generated volume consisting of 150,000 tetrahedra. This was rather large at that time and hence an Intel Paragon XP/S with 128 processors still required 115 seconds to render a single image.

Parker et al. [Parker99b] presented an interactive iso-surface renderer for regular grids (see Section 4.1), and used the same acceleration structure as for unstructured tetrahedral meshes. To this end, the multi-level grid requires only a slight modification at each leaf. Each leaf now stores a list of tetrahedra spatially located in the grid cell associated with the leaf. Ray traversal then starts as described for the rectilinear case. At leaf level, each of the tetrahedra found in that cell is then sequentially tested against the ray for intersection. All tetrahedra are treated as independent items, which avoids storing connectivity information. Similar to the regular grid approach, the iso-surface can again be implicitly computed using this time local barycentric coordinates. This intersection computation is even easier, since the iso-surface is

planar by definition (triangle or quadrangle) within each tetrahedron leading to a set of four linear equations to be solved. The system performance was reported as 11 fps for a bioelectric field simulation of approximately one million tetrahedra using 124 processors on an SGI Reality Monster.

The latest development in CPU ray tracing of tetrahedral grids uses bounding volume hierarchies (BVH) for fast iso-surface rendering, and supports time-dependent data sets [Wald07]. The BVH is constructed over the tetrahedral grid by merging together tetrahedra with similar ranges of scalar values. As for the implicit kd-tree, each bounding volume in the hierarchy is provided with minimum and maximum values representing the range of scalar values. Further speedup is gained from packet traversal and frustum culling. Mid-sized data sets (e.g. orbital, bucky-ball, Blunt-fin) consisting of 150.000 to 220.000 triangles can be rendered at 40 fps on a desktop PC using a 512^2 screen resolution. However, this approach is not only restricted to iso-surface rendering but to tetrahedral meshes also.

In summary, this section presented approaches either restricted to tetrahedral meshes [Garrity90, Ma95, Wald07] or curvilinear grids [Wilhelms90, Frühauf94]. The latest approaches [Parker99b, Wald07] demonstrate the use of an acceleration structure when restricted to certain visualization tasks. The upcoming section will present a fast alternative to the ray-plane intersection, namely the computational space traversal, based on *Plücker coordinates* [Marmitt05, Marmitt06c, Marmitt06b].

6.2 Theoretical Background

This section explains the theoretical background for *Plücker coordinates* and bilinear patch intersections before describing both traversal algorithms in detail.

6.2.1 Plücker Coordinates

Plücker coordinates provide a way of specifying directed lines in three-dimensional space [Erickson97]. Basically, these coordinates represent a ray as an *oriented line*. Suppose a ray $r(t) = o + dt$ is given, this results in the following six-vector:

$$\pi_r = \{d : d \times o\} = \{p_r : q_r\}. \tag{6.1}$$

Plücker coordinates are homogeneous. Multiplying all six coordinates by any real number results in new *Plücker coordinates* for the same line.

Additionally, coordinates given by $\pi_r = (p_x, p_y, p_z; q_x, q_y, q_z)$ always satisfy the following equation:

$$p_x * q_z - p_y * q_y + p_z * q_x = 0. \tag{6.2}$$

Now, two given oriented lines, r and s, in space can interact in three different ways: r might intersect s, r might pass counter-clockwise about s, or r might pass clockwise about s. This information is encoded in a *permuted inner product* of *Plücker coordinates*, which is rather easy to compute. For two lines r and s, represented by *Plücker coordinates*, this results in

$$\pi_r \odot \pi_s = p_r \cdot q_s + q_r \cdot p_s. \tag{6.3}$$

A positive result means that r passes counter-clockwise about s, while in the negative case, r passes clockwise about s. If this product is zero, the lines intersect (see Figure 6.2). Care must be taken about the direction, i.e. the *Plücker product* for a line $r \rightarrow s$ differ from $s \rightarrow r$. The sign of the permuted inner product changes.

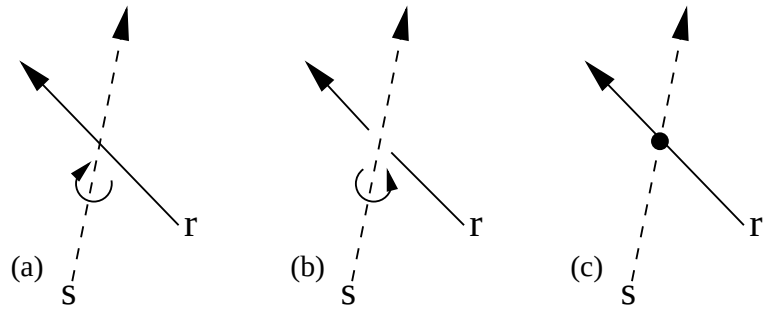


Figure 6.2: Three possible cases for the Plücker test: Looking toward the direction of s, (a) ray r passes counter-clockwise about line s, (b) ray r passes clockwise about line s, and (c) ray r intersects line s.

The geometric interpretation is therefore simple to follow. $\pi_r \odot \pi_s$ is nothing more but the signed volume of a tetrahedron spanned by the (end points of) lines r and s. This is no surprise since the Equation 6.3 calculated the determinant of the tetrahedron specified by the four points. *Plücker coordinates* are simply a special case for three dimensions. For n dimensions the discussed properties are referred to as Grassmann coordinates.

6.2.2 Bilinear Patches

A bilinear patch is determined by a set of four points (p_0, p_1, p_2, p_3) in three-dimensional space, which are not necessarily coplanar. If one wants to find the intersection of a ray $r(t) = o + dt$ with a bilinear patch, the following equation must be solved:

$$\begin{aligned} o + dt &= (1-u)(1-v)p_0 + (1-u)vp_1 \\ &\quad +u(1-v)p_2 + uvp_3, \end{aligned} \tag{6.4}$$

where $(u, v) \in [0, 1]^2$, i.e. the contribution of each point is described as a weighting of two parameters (u, v). Substituting the ray equation into Equation 6.4 leads to a quadratic equation, since two intersections are possible. This unfortunately introduces a costly square root operation. The nearest intersection is then to be returned. It is straightforward to solve the equation for u and v. Numerical instabilities due to small denominators can be avoided by choosing the largest absolute value of the denominators. A fast and robust implementation was described by Ramsey et al. [Ramsey04], which was also used here.

After given this theoretical background, two incremental traversal algorithms for both tetrahedral meshes and hexahedral grids are described, where the traversal decision is made using *Plücker coordinates*. Bilinear patches will be used in the following section where hexahedral primitives are traversed, since they provide a better parameterization for hexahedral faces. Both rely on kd-tree for finding the initial (i.e. first) primitive along a ray.

6.3 Tetrahedral Meshes

Many scientific areas require an unstructured point cloud, e.g. for simulating physical processes. However, from the visualization perspective, it is rather tedious to traverse point clouds; therefore, such data sets are often enhanced with a tetrahedral mesh providing an additional topology. This can be achieved by a Delaunay-tetrahedralization [Choi02], which is also assumed for the approach explained below.

In such a mesh, each tetrahedron consists of four *vertices* and six lines connecting them. Each face of a tetrahedron is either shared between two adjacent tetrahedra (i.e. *inner faces*), or they belong to the boundary of the tetrahedral mesh (i.e. *boundary faces*). Sliding interfaces, i.e. tetrahedra that share only a part of a face, are not allowed. As Platis and Theoharis [Platis03]

showed, *Plücker coordinates* can be used to quickly determine, which faces are intersected by a ray piercing a tetrahedron.

In the following, this method is extended to allow for traversal of a tetrahedral mesh. The first step is to determine the initial (i.e. first) tetrahedron along a ray path. This can be found using a well-known acceleration structure for ray tracing, i.e. kd-trees on the boundary faces of the tetrahedral mesh. Once the first tetrahedron is known, all subsequent tetrahedra are traversed using *Plücker coordinates*. In each cell either iso-surface rendering or an emission-absorption model is applied.

6.3.1 Finding the Initial Tetrahedron

To find the initial tetrahedron along a ray, the first step is to extract all boundary faces during a preprocessing step (see Figure 6.3). It is easy to see that this is the set of all tetrahedral faces not shared with any other tetrahedra in the set. A kd-tree can then be used as an acceleration structure, which has been proven as a fast and efficient technique when using ray tracing [Havran01]. The implementation suggested by Wald et al. [Wald04a] was used especially for this purpose. The data associated with the returned triangle provides the tetrahedron as well as its entry face so that the incremental Plücker traversal can be used thereafter.

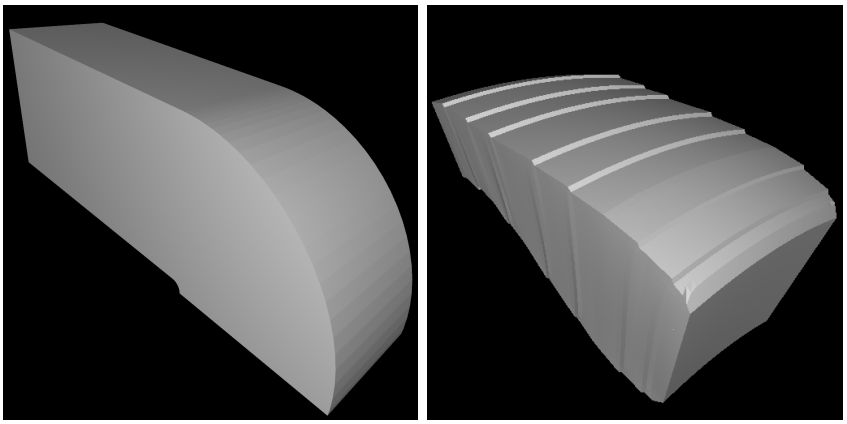


Figure 6.3: Left: *Boundary representation of the Blunt-fin data set. This is identical for the tetrahedral and hexahedral cases, since all boundary faces are decomposed into triangles.* Right: *Boundary representation of the Combustion Chamber consisting of hexahedral primitives. Each hexahedral boundary face is decomposed into two triangles.*

6.3.2 Mesh Traversal in Plücker Space

Here, the ray-tetrahedron intersection algorithm introduced by Platis and Theoharis [Platis03] is extended for the traversal of tetrahedral meshes. They specifically exploited the fact that each tetrahedron can be decomposed into four triangles. To find the exiting face, each triangle must be checked for intersection with the ray. This can be achieved by converting all edges and the intersecting ray into *Plücker coordinates*. Using the properties described in Section 6.2, a ray intersects the triangle if and only if all results have the same sign. The only condition that must be checked is that the *Plücker coordinates* are either all clockwise or all counter-clockwise. Figure 6.4 illustrates the basic idea.

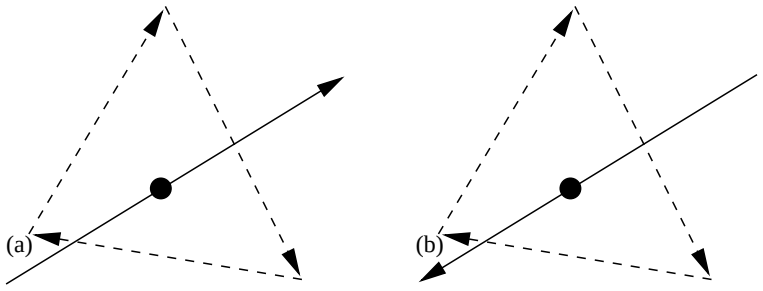


Figure 6.4: Two different configurations with the same (positive) intersection result: (a) the ray passes clockwise about all line segments of the triangle, hence all signs are positive, and (b) the ray passes about counter-clockwise all line segments, and therefore all signs are negative.

A straightforward approach would be to check all three possible exit faces separately (see Figure 6.5). The resulting number of tests varies in this case between three and nine, depending on which triangle returns a positive intersection first. Empirical investigations show that the average number of tests lead to approximately 4.9 tests per tetrahedron. In a tetrahedral mesh, however, the *Plücker test* can be even more efficiently applied, since all three results (i.e. $v_0 \rightarrow v_1$, $v_1 \rightarrow v_2$, and $v_2 \rightarrow v_0$) from the shared face can be re-used. It suffices then to test the edges $v_0 \rightarrow v_3$ and $v_1 \rightarrow v_3$ against the incident ray. If the sign of these two tests differ, edge $v_2 \rightarrow v_3$ must be checked as well. An important premise is that all interior faces are shared by exactly two tetrahedra.

Applying the above mentioned optimizations, the number of tests drop to 2.67 on average while the performance increases to roughly 17 million processed tetrahedra per second. Also, the raw performance of tetrahedra

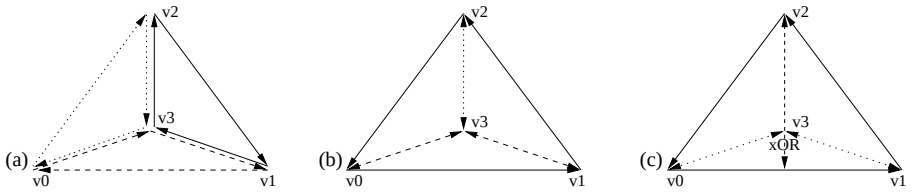


Figure 6.5: (a) Naïve approach: All three exiting faces of the triangle are tested independently although each line is shared by two faces (b) Optimized approach: The solid line tests are given from the previous tetrahedron and the dotted line need only be computed if the test with the dashed lines failed. The direction of each line can be reversed by negating the sign of the Plücker test*. (c) Theoretical optimum: The first dashed line divides the possible exit space into two halves, where one additional test (one of the dotted lines) suffices to determine the exit face.*

processed per second increases by 55% on average compared to the naïve approach (see Table 6.1). Another advantage is that only one vertex coordinate and the corresponding scalar value have to be fetched per tetrahedron. This keeps the memory bandwidth low and improves overall performance due to better cache usage. Once the exit face is identified, connectivity information helps to determine the neighboring tetrahedron and requires only 16 additional bytes per tetrahedron.[3]

	naïve approach		optimized approach	
Dataset	#π-tests	Mt/s	#π-tests	Mt/s
Blunt-fin	4.714	9.346	2.669	17.106
Orbital	4.976	9.615	2.675	17.101
Bucky-ball	5.373	9.615	2.663	17.612

Table 6.1: Comparing the number of Plücker tests and the rendering rate (millions tetrahedra per second) for the naïve and the optimized approaches. The optimized approach reduces the number of tests, resulting in a performance increase of 70% (3 GHz Athlon64, 2 GB memory).

The minimum number of *Plücker tests* per tetrahedron in a mesh is 2.33 in average, as Figure 6.5 (c) shows. The first test with the dashed line divides the possible exit face into two halves, reducing the problem of finding the exit

[3]In contrast, Weiler et al. [Weiler03] reported that 160 bytes per tetrahedron were necessary to use their approach.

face from three to two faces. Another test with either of the dotted lines reveals the correct exiting face. For now, even two *Plücker tests* would suffice. However, to compute barycentric coordinates to interpolate the scalar value at the face, a third test must be applied in $\frac{1}{3}$ of all cases on average, leading to 2.33 tests per tetrahedron [Marmitt06c]. Unfortunately this introduces additional branches, as pseudo-code listings of both approaches show, destroying the advantage gained by using today's highly pipelined processors, as shown in Algorithm 9.

Algorithm 9 Determining the exit face within a tetrahedron: implemented approach (left) versus theoretical optimum (right).

```
while New Tetrahedron do
  if π(v0, v3) < 0 ∧ π(v1, v3) > 0
  then
    {Exit with Face 2}
  else if π(v2, v3) > 0 then
    {Exit with Face 0}
  else
    {Exit with Face 1}
  end if
end while
```

```
while New Tetrahedron do
  if π(v2, v3) < 0 then
    if π(v0, v3) > 0 then
      {Exit with Face 1}
    else
      {Exit with Face 2}
    end if
  else if π(v1, v3) < 0 then
    {Exit with Face 0}
  else
    {Exit with Face 2}
  end if
end while
```

Moving from one tetrahedron to the next, the algorithm needs to know which of the four faces is shared with the tetrahedron just visited. To this end, each *face-id* is set to the id of the vertex lying "across" the face. In other words, it is the id of the vertex which is not in the set of the three vertices defining this face. For example, the face determined by the vertices v_0, v_1, v_3 has the (local) *face-id* $v_2 = 2$. Since the subsequent tetrahedron receive the same *face-id*s the only missing vertex for processing this tetrahedron can be fetched immediately by using the (already known) *face-id* of the shared face, which works well with the new implementation.

Of course, the performance gain is less significant when taking the entire algorithm into account. Finding the initial tetrahedron and computing the shading also have a large impact. A detailed analysis shows that identifying the exit face consumes about one third of the total computational cost. This leads to an expected performance gain of at most 35%. It is actually a little less, since the optimized method introduces additional overhead. This is due to the fact that the system must know, which vertices of both tetrahedra are

shared. The sequence of the vertices is also important since the *Plücker test* is direction sensitive. For the same reason the *face-id* of the exit face needs to be checked using two additional conditionals.

The data parallel scheme requiring the calculation of up to three *Plücker tests* per tetrahedron suggests the use of SIMD extensions, e.g. via the SSE instruction set [Intel], which would allow the calculation of all *Plücker coordinates* simultaneously. However, arranging the data appropriately for the SSE registers results in significant overhead. Additionally, full parallelism cannot be achieved for the four-component wide SSE instruction as there are only three parallel *Plücker tests* computable. This results in very little gain using SIMD computations and in this case, led roughly to the same performance.

6.3.3 Iso-Surface Cell Intersection

For all rendering tasks, the interpolated value at the entry and exit points must be computed first. *Plücker products* provide the scaled barycentric coordinates, which is a major advantage compared to plane-intersection approaches. Thus, each *Plücker product* need only be divided by the sum of all three products associated with the tested face:

$$w_i = \pi_r \odot \pi_{e_i} \qquad \text{and} \qquad u_i = w_i / \sum_{j=0}^{3} w_j. \tag{6.5}$$

For the emission-absorption model, these values can either be used directly for accumulation, or an additional super-sampling can be applied since evaluating at the tetrahedral faces only does not necessarily obey the Nyquist limit. Since the reconstructed signal along the ray is described using linear equations within a tetrahedron, this is not necessary for maximum intensity projection. Recall that maximum intensity projection saves the highest interpolated value found along the ray path, which is then used as the final pixel color.

When using iso-surface rendering, the reconstructed signal along the ray is searched for a specific value, i.e. the iso-value. This is usually achieved by reconstructing the signal piecewise per tetrahedron. Once a tetrahedron along the ray is detected, where the iso-value is located within the range determined by the entry and exit points, a linear interpolation returns the intersection with the iso-surface. As depicted in Figure 6.6, the four vertices of a tetrahedron are treated as a linear equation, which always results in a planar iso-surface (either a triangle or a quadrangle). Iso-surface rendering, like maximum intensity projection, does not have to obey the Nyquist limit.

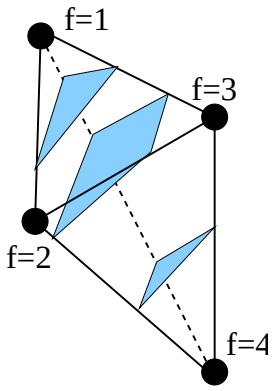


Figure 6.6: A tetrahedron with associated function values at all four vertices. Possible iso-surfaces (blue) are shown for $f = 1.8$ (triangle), $f = 2.5$ (quad), and $f = 3.7$ (triangle). Values outside the range, e.g. $f = 0$, produce no iso-surface.

Solving this system requires only simple mathematics. Given a tetrahedron with vertices v_i and corresponding scalar values s_i ($i \in \{0, 1, 2, 3\}$), the linear function is calculated by solving the system of four equations:

$$s_i = ax_i + by_i + cz_i + d \tag{6.6}$$

with $v_i = (x_i, y_i, z_i)$ for the unknowns a, b, c and d. The intersection is then found by substituting the ray equation into (6.6) and setting s_i to the chosen iso-value. Figure 6.7 illustrates the results for the described rendering modes as well as the application of a transfer function to highlight regions of interest.

The results in Table 6.2 show that the performance of the pure traversal is about 10 million tetrahedra per second. Note that the entire implementation relies on the CPU. No special hardware is needed for the implementation. Iso-surface rendering requires, as pointed out earlier, additional computations, i.e. calculating barycentric coordinates and applying a linear interpolation. These two operations lead to a performance drop by 50%, but still achieves over 5.3 million intersections per second. The number of processed tetrahedra per second is even a little slower for semi-transparent rendering, which is caused by tracking the emission and absorption values. More importantly in both cases the number of processed tetrahedra per second is stable, which suggests that other data sets of size will lead to similar rendering performance.

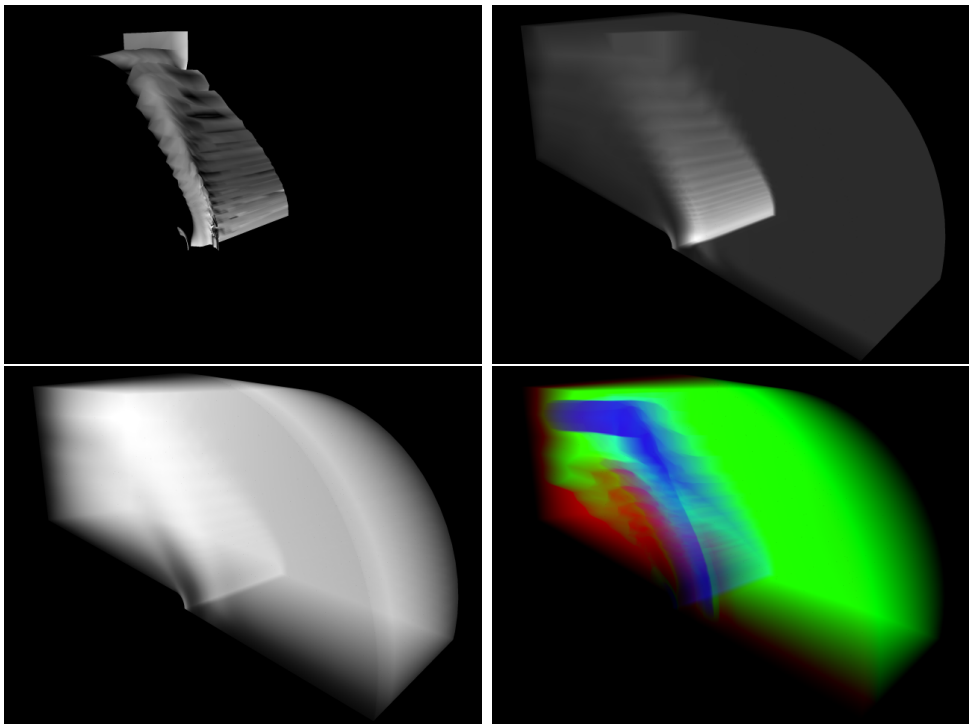


Figure 6.7: Upper Left: *The tetrahedral* Blunt-fin *data set rendered as an iso-surface,* Upper-right: *maximum-intensity-projection,* Lower-left: *semi-transparent rendering, and* Lower-right: *with transfer functions.*

6.3.4 Gradient Computation

From the previous observations, it follows directly that the orientation of the plane describing the iso-surface is independent of the iso-value; hence, the plane normal $N_t = (a\ b\ c)^T$ is constant within the cell. Unfortunately, this results in discontinuities at the tetrahedra surfaces, which decreases the rendering quality significantly (see Figure 6.8). For better results, one can calculate a normal N_v per vertex of the tetrahedron. This can be expressed as the sum of all n tetrahedra h_t connected to this vertex and weighted with volume $V(h_t)$:

$$N_v(a\ b\ c)^T = \sum_{t=0}^{n} N_t(a\ b\ c)^T * V(h_t) \tag{6.7}$$

These vertex normals can be pre-computed and attached to the vertices of the data set. In a final step, 3D barycentric coordinates (i.e. for a tetrahedron) of the intersection point within the tetrahedra are calculated

Dataset	# Tetrahedra	T-Traversal	iso-surface	semi-transparent
Blunt-fin	224,874	11.175	5.394	4.189
Orbital	148,955	10.568	5.379	4.020
Bucky-ball	176,856	10.514	5.237	4.049

Table 6.2: Number of tetrahedra processed (in millions per second) for the pure tetrahedra traversal, for iso-surface rendering, and for semi-transparent rendering. Although the number of tetrahedra differ significantly, the number of processed tetrahedra per second is stable.[5]

and weighted with the associated (pre-computed) vertex normal to obtain a smooth iso-surface normal. Finally, the vector resulting from Equation 6.7 must also be normalized.

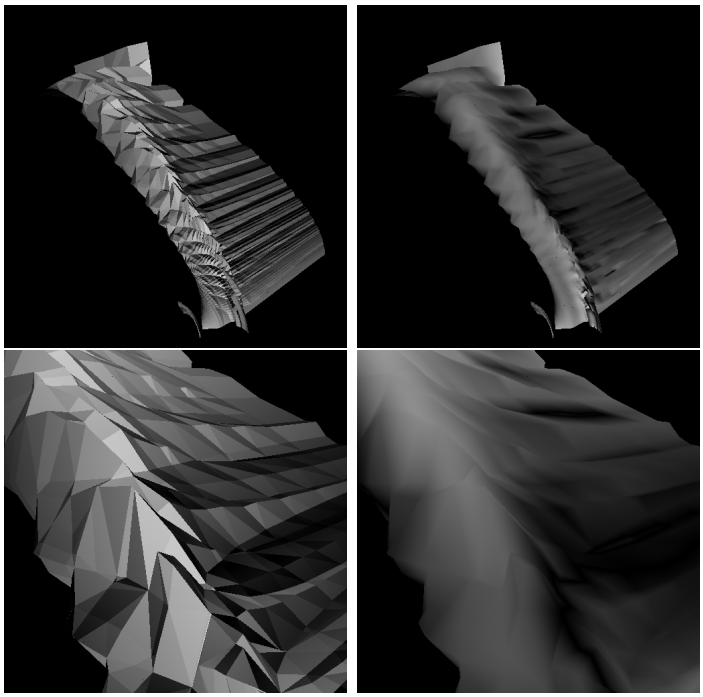


Figure 6.8: Overview (top) and close-up views (bottom) of both normal calculations. Left: *Normals per tetrahedron result in severe artifacts while,* Right: *weighted vertex normals leads to smooth color transitions.*

6.3.5 Memory Requirements

Compared to regular grids, the data structure for tetrahedral meshes is rather complex (see Figure 6.9). First, it is necessary to store not only the scalar value, but the spatial position of each vertex, which results in additional 12 bytes per vertex when using 32-bit floating point precision. Since every point is used for several tetrahedra, it is not wise to attach this information directly to each tetrahedron. The usual alternative is to save four index pointers, leading to 16 bytes of memory consumption per tetrahedron.

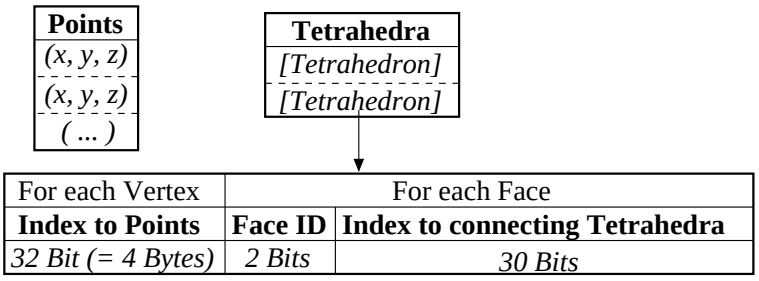


Figure 6.9: The data structure for the tetrahedral traversal algorithm consists of two tables. Table Points *contains all 3D vertices. Table* Tetrahedra *holds entries for the* Tetrahedron *structure. Each* Tetrahedron *structure have four indices for the vertices as well as a* face-id *and a connection-id.*

Another 16 bytes are needed for neighbor connectivity information. If the index pointer is restricted to 30 bits the remaining two bits can be used for storing the *face-id* of the connecting tetrahedron in the case of an inner face. This avoids checking the *face-id* with costly conditional statements while traversing the tetrahedron.

Assuming 32-bit floating point scalars, the memory consumption is 16 bytes per vertex and 32 bytes per tetrahedron, which is four times less than Weilers approach [Weiler03] requiring 160 bytes per tetrahedron. Further space consumption depends on whether and where to store the gradient vector. As was shown in Section 6.3.4, the gradient vector can be either attached to the tetrahedron or to the vertex.

6.3.6 Scalability Measurements

Scalability measurements are given using the proven shared-memory system (eight dual-core 2 GHz Opteron 870s, 64 GB RAM). The following tables present, if not stated otherwise, an average mean of four differently chosen viewpoints. Unlike regular grids, the average number of tetrahedra along a

ray path largely depends on the viewpoint. Therefore three viewpoints were orthogonally chosen with respect to one of the three spatial dimensions along as well as an additional typical perspective view.

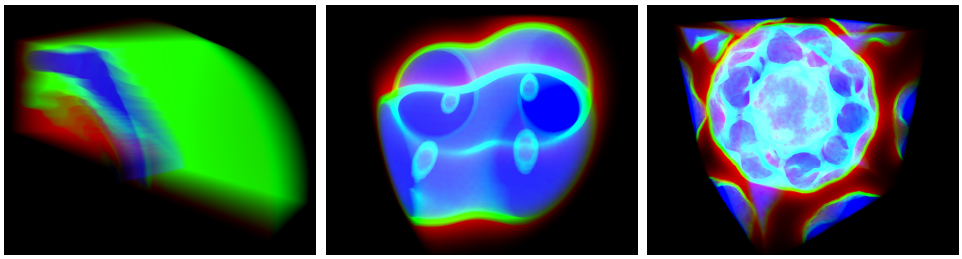


Figure 6.10: Unstructured data sets: Left: *Blunt-fin data set consisting of 224,873 tetrahedra,* Middle: *Orbital with 148,955 tetrahedra, and* Right: *Bucky-ball with 176,856 tetrahedra.*

Three well-known unstructured data sets were evaluated from different viewpoints using iso-surface rendering and semi-transparent rendering: the *Blunt-fin* data set with 224,873 tetrahedra, the *Orbital* data with 148,955 tetrahedra, and the Buckminster fullerene (*Bucky-ball*) containing 176,856 tetrahedra. Rendered sample images of all three data sets can be seen in Figure 6.10. A detailed description of the testing system, data sets, and conditions is provided in Section 6.5.

		Number of Cores (fps)					
Data Set	Screen Res	1	2	4	8	12	16
Blunt fin	512^2	1.13	2.29	4.59	9.32	12.86	15.78
	1024^2	0.29	2.32	1.17	2.34	3.47	4.61
Orbital	512^2	0.86	1.72	3.46	6.91	10.34	13.86
	1024^2	0.22	0.44	0.87	1.75	2.61	3.34
Bucky-ball	512^2	0.77	1.55	3.10	6.15	9.24	12.04
	1024^2	0.20	0.39	0.78	1.57	2.33	2.87

Table 6.3: Distributing the rays shot among several nodes allow for interactive rendering performance with 4 to 16 nodes since this approach has an inherent linear scalability. All figures represent average fps from four different viewpoints.

In general, it is not recommended to render iso-surfaces with an incremental traversal, since almost in every case it is better to add some sort of hierarchy to the volume topology to speedup cell processing (e.g. [Parker99b,

Wald07]). It is nevertheless interesting to see the performance gain compared to semi-transparent rendering.

Table 6.3 shows the expected linear scaling while increasing the number of rendering cores of the shared memory system. Due to a different implementation of the rendering system, each node keeps its own copy of the data set and other data structures. As expected, the performance is not suitable for interactive applications, especially for larger screen resolutions. To render iso-surfaces with this implementation, at least two dual-core processors are needed for a 512^2 viewport, and all eight dual-core processors for a 1024^2 viewport.

It is especially important to test the performance for semi-transparent rendering since this is the main application for an incremental traverser. Visiting all primitives along a ray is slow, but allows for a more accurate representation compared to the approach discussed in Section 4.5, since no approximation for homogeneous regions is used.

		Number of Cores (fps)					
Data Set	Screen Res	1	2	4	8	12	16
Blunt fin	512^2	0.81	1.62	3.25	6.47	9.49	11.69
	1024^2	0.20	0.40	0.81	1.64	2.44	3.27
Orbital	512^2	0.59	1.21	2.45	4.92	7.32	9.68
	1024^2	0.16	0.31	0.62	1.24	1.86	2.44
Bucky-ball	512^2	0.72	1.48	2.98	5.95	8.90	11.94
	1024^2	0.19	0.38	0.76	1.52	2.26	2.96

Table 6.4: Compared to iso-surface rendering, performance is again significantly reduced. This is caused by the increased number of primitives along a ray, which is on average 30% higher. All figures are average fps from different viewpoints.

Table 6.4 shows the overall rendering performance in fps for semi-transparent rendering. The average number of primitives processed increases by approximately 30%, since the traversal does not stop as soon as a iso-surface is found but when the absorption reaches 100 %. Note that the frame-rate drops by approximately the same amount. This is due to the fact that the number of *Plücker tests*, and hence the number of tetrahedra processed per second, is stable for this algorithm.

6.4 Hexahedral Grids

Hexahedral grids are better known as curvilinear data sets. All vertices are represented in a 3D grid, but the distance between two neighboring points varies. Ideally, a bijective function is supplied, which allows the conversion of points from physical space to computational space, and vice versa. In reality, this function is often not available. Therefore it is assumed that only scalar values, together with the 3D grid vertices, are given.

6.4.1 Finding the Initial Hexahedron

Again, the initial hexahedron along a ray is found using the same kd-tree as described in Section 6.3.1. All boundary faces of the data set are extracted in a preprocessing step, whereas each (boundary) quadrilateral (i.e. hexahedral face) is decomposed into two triangles (see Figure 6.3). This makes the kd-tree traversal as efficient as for the tetrahedral traversal.

6.4.2 Grid Traversal

Two approaches are evaluated in the following subsections. First, the *Plücker test* introduced in Section 6.3.2 can be easily extended to traverse hexahedral grids, too. This is possible under the assumption that all faces of a hexahedra are planar. However, it is still unclear whether the *Plücker test* can be used to derive parametric coordinates for hexahedral faces. Hence, in this approach each face needs to be decomposed into two triangles for calculating the coordinates of interpolating the scalar values. This leads to parametric discontinuities along the additional diagonal, but allows the intersected faces to be determined unambiguously. The second method extends this approach using bilinear patches. Here, no discontinuities occur but numerical issues can sometimes lead to inconsistent decisions for rays at the edges (see Section 6.2.2).

No extra memory for hexahedral adjacency is required since this is implicitly encoded into the grid. Once the exit face has been determined, only an index pointer is modified by incrementing or decrementing with respect to the dimension. If this pointer exits the volume bounds, the traversal stops.

6.4.2.1 Plücker Space

Applying the same optimizations used for the tetrahedral traversal from Section 6.3.2 lead to at most four tests to decide, which one of the five possible faces is the exit face. This algorithm is designed for convex hexahedral faces

since concave faces do not occur in most simulations. Figure 6.11 illustrates the algorithm. Suppose that the entry face is given by the vertices v_0, v_1, v_2, and v_3. The opposite face is determined by v_4, v_5, v_6, and v_7.

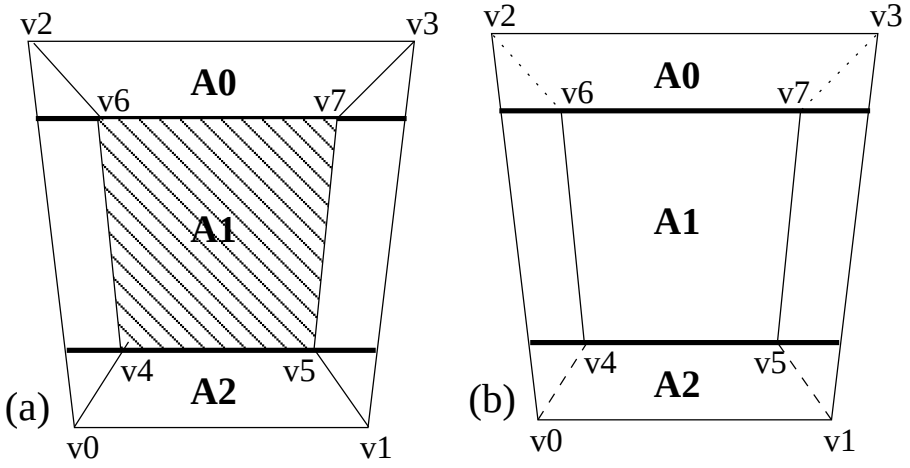


Figure 6.11: a) To determine the exiting face, the hexahedron is subdivided into three areas. (b) The next step is then to check which face is intersected by applying up to two additional Plücker tests (A_0: dotted edges, A_1: solid edges, and A_2: dashed edges).

In the first step, the *Plücker test* is performed with the edges $v_4 \rightarrow v_5$ and $v_6 \rightarrow v_7$. This results in three areas, A_0, A_1, and A_2. Note that each area contains at most three faces of the hexahedron, i.e. there are only three faces left to check. To accomplish this, each area is treated differently, i.e. the edges $v_2 \rightarrow v_6$ and $v_3 \rightarrow v_7$ for A_0, $v_4 \rightarrow v_6$ and $v_5 \rightarrow v_7$ for A_1, and $v_0 \rightarrow v_4$ and $v_1 \rightarrow v_5$ for A_2 are tested first. The second test needs only be performed if the first does not lead to a decision. This second step is illustrated in Figure 6.11(b). Now simple sign comparisons are sufficient to determine the correct exit face.

Fetching the data for the next cell and saving the appropriate data for reusing requires more time compared to the tetrahedral mesh. Analysis shows that the costs for these operations are seven times higher compared to the previously discussed tetrahedral traversal, where only one vertex and scalar value need to be fetched in each step.

Extending this algorithm to support concave faces is unfortunately difficult to achieve. Figure 6.12(left) shows one important problem arising with this extension: line $v_4 \rightarrow v_5$ cannot be used for a sign test, since for parts of this segment (colored area), this test fails. One is faced with similar prob-

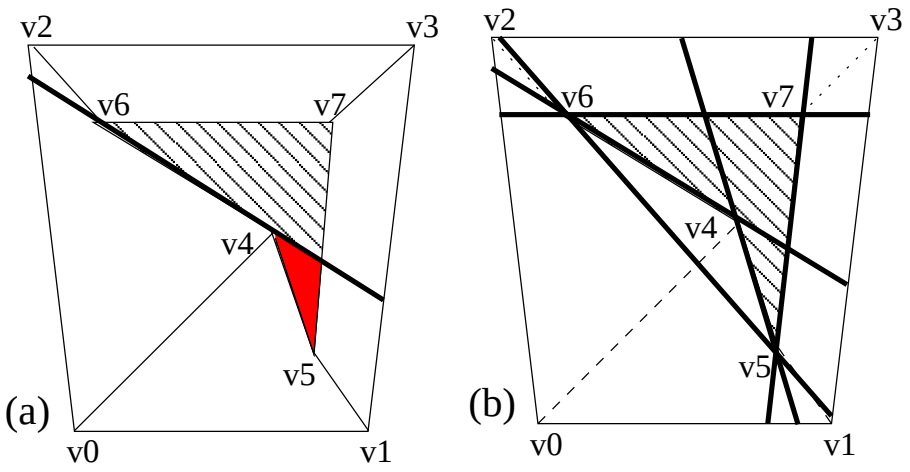


Figure 6.12: (a) The concave face illustrated cannot be handled correctly with the approach discussed previously, since it is impossible to test line $v_4 \rightarrow v_5$ appropriately with respect to the colored area. (b) As a possible solution, the convex hull of the concave face is partitioned into triangles, which can then be handled separately. This includes two additional calculations of Plücker coordinates and a test for concave polygons.

lems when trying to identify other areas uniquely with this test. A possible solution is outlined in Figure 6.12(right). Constructing the convex hull of this sample concave face yields a triangle, which can then be tested with *Plücker coordinates* again. This leads to a total of six tests including the additional diagonal. Although the overhead introduced probably lies below 20%, an additional test for detecting concave faces must be implemented as well. Possible techniques are ray-crossing counts used by Hong et al. [Hong99] or the winding number test (see [Haines94] for details on both methods).

6.4.2.2 Bilinear Patch Extension

As it will be shown in the next section, one runs into problems when trying to derive parametric coordinates from the face using *Plücker tests*. This can be avoided by considering bilinear patches as an alternative. Note that it is even possible to traverse the curvilinear data set directly by applying this test to each of the five possible exit faces, and to stop as soon as a patch is hit, i.e. $(u, v) \in [0, 1]^2$. Due to the high computational cost of bilinear patch intersections, this is, however, undesirable. Furthermore, in some configurations numerical imprecision can lead to inconsistent decisions at edges. In particular, an edge of a bilinear patch depends on all four

face vertices. In contrast, *Plücker coordinates* allow for a unique decision whether a ray passes clockwise or counter-clockwise about a line in space since they depend solely on the two edge vertices. It is important to handle this inconsistency so that a ray can traverse the correct cells within a volume.

Considering these facts, a combination of *Plücker coordinates* and bilinear patches seems to be the optimal solution [Marmitt06c]. The first step is again to compute *Plücker coordinates* of the lines $v_4 \rightarrow v_5$, $v_6 \rightarrow v_7$, and the ray, which results in three areas, A_0, A_1, and A_2 (see Figure 6.11). In contrast to the previous method, in each area all three possible exit faces are checked using the bilinear patch intersection. As a result, raw traversal performance is 15% lower compared to the pure Plücker traversal. However, time is saved in the overall performance since the surface parameters (u, v) are already known.

Finding the exit face by using bilinear patches is rather simple. The patch coordinates are computed for each possible exit face within a given cell, stopping as soon as a patch with $(u, v) \in [0, 1]^2$ is found. This means that at most three faces need be checked per hexahedron. Assuming that the face opposite of the entry face has a higher probability than the other four faces due to its (apparent) larger area, the process runs faster when testing it first. This is of course not in true all cases since it depends on the individual shape of a hexahedron. On the other hand, giving preference to the opposite face requires hardly more time and is therefore at least as fast as testing the faces in an arbitrary order.

One drawback of this algorithm is that it introduces numerical inconsistencies at the edges, since the two bilinear patches sharing this edge claim that the ray is intersecting their face. Such inconsistencies are caused by numerical issues (i.e. insufficient floating point accuracy), where an additional *Plücker test*, equivalent to the second step described in Section 6.4.2.1, solve this problem. These numerical issues depend on the spatial extension of the cells and are therefore data-dependent. However, the additional *Plücker tests* in both tested data sets and adds only a negligible performance impact.

6.4.3 Iso-Surface Cell Intersection

The previous section pointed out that *Plücker coordinates* generally *cannot* be used to derive parametric coordinates from a hexahedral face. For this reason, bilinear patches were added to the traversal. Figure 6.13 sketches, why this not easy and therefore probably computationally expensive. Let a, b, c, and d denote the result of the *Plücker test* for the intersecting ray with each of the four edges. One might think that a surface parameterization is still possible by scaling all four values. However, it still remains unclear how

to scale the values properly. Simply dividing each tetrahedral volume by the pyramidal volume obviously does not work. The base area of a pyramid is quadrilateral and thus it is hard to derive meaningful scaling vectors, and hence parametric coordinates for interpolation, with the *Plücker test*.

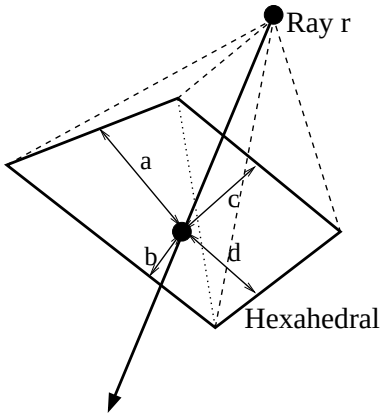


Figure 6.13: This figure illustrates why the Plücker test *fails to provide meaningful parametric coordinates in the case of a hexahedral face. The scaling factor for the* Plücker tests *a and b differs from the one for c and d, since they were calculated using two different tetrahedral volumes.*

Instead of searching for an analytical solution which is probably too costly to compute, each face is simply split into two triangles so that barycentric coordinates can again be computed. Depending on the previously computed *Plücker tests*, one or two additional tests are required, which decreases performance significantly [Marmitt06c]. Furthermore, decomposing a face into two triangles leads to unwanted discontinuities if the values of the hexahedron are poorly distributed (see Figure 6.14).

In addition to the traversal operation, interpolation of the scalar values at these points is also performed for shading the volume. This introduces one or two additional *Plücker tests* depending on what can be re-used from the exit face decision. When using bilinear patches, the parametric coordinates are already known, and hence only the interpolated scalar values at the entry and exit faces must be computed. Discontinuities are completely avoided in this way.

Applying iso-surface rendering is now fairly simple. If the user-defined value is within the values interpolated at the entry and exit faces, an additional linear interpolation determines the intersection with the implicit surface. Of course, it would be better to trilinearly interpolate in computational

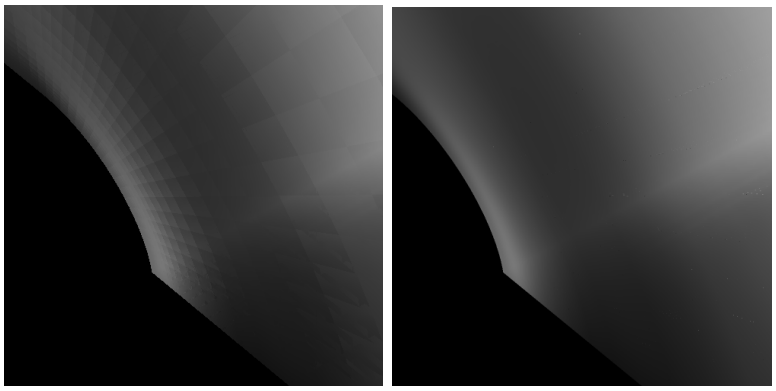


Figure 6.14: Artifacts when rendering the Blunt-fin data set. Left: *Decomposing produces artifacts, while* Right: *the bilinear patch delivers smooth results.*

space and derive a cubic function. This additional computations would, however, significantly decrease the performance. For semi-transparent rendering or maximum intensity projection the interpolated values are used directly. It is also possible to super-sample these values in the computational domain to obtain smoother results.

Dataset	H-Marching		iso-surface		semi-transparent	
	Plücker	Hybrid	Plücker	Hybrid	Plücker	Hybrid
Blunt-fin	7.49	7.58	3.15	5.46	2.21	3.17
Combustion	8.44	8.83	2.83	4.49	2.12	3.45

Table 6.5: Number of hexahedra processed (in millions per second) for the pure hexahedra traversal; for iso-surface rendering, and for semi-transparent rendering. The hybrid approach always performs better than the pure Plücker traversal. Far better performance results from the fact that bilinear patches already include parameterization for face interpolation (3 GHz Athlon64, 2 GB memory).

The results in Table 6.5 show that the performance for the pure traversal is still about eight million hexahedra per second. When traversing the hexahedral mesh, both approaches show similar performance. In both cases, this performance is halved when the data set is actually rendered. Pure Plücker traversal needs an additional *Plücker test* for face interpolation, which is already included in the hybrid approach. Therefore the hybrid approach can process far more hexahedra for both iso-surface and semi-transparent render-

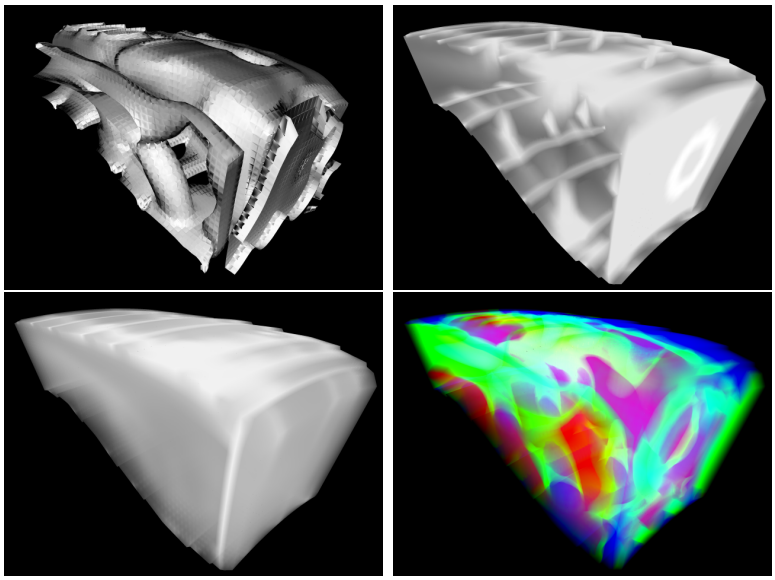


Figure 6.15: Various renderings of the Combustion Chamber data set. Upper-left: *rendered as iso-surface,* upper-right: *maximum intensity projection,* lower-left: *semi-transparent rendering, and* lower-right: *with transfer functions.*

ing. Further analysis shows that the number of processed hexahedra varies about 20% among the three tested data sets. This is caused by bilinear patch computation, which is much faster in the case of small denominators since an approximation is used in this case without costly square root operations.

6.4.4 Gradient Computation

Since it was decided before to avoid the costly transformation into computational space for finding the iso-surface, it is now also for the gradient computation necessary to use a method other than trilinear interpolation. Let us assume for a moment that pre-computed normal vectors are available at each vertex. To compute the normal at the intersection of the ray and the iso-surface, it is then possible to re-use the same parametric coordinates used for interpolating the scalar values, except that now a vector is interpolated at the entry and exit face. Another linear interpolation between the gradient vectors at the entry and exit faces based on their distances to the iso-surface, yields the normal.

For computing the vertex normals of a hexahedron, a method suggested by Frühauf [Frühauf94] is applied [Marmitt06c]. Note that in [Frühauf94] this

method is used for ray traversal and not for any gradient computation. In fact, in general the gradient cannot be transformed with the same matrix as the object points. In this case, the following approximation works, however, well enough.

The basic idea is to compute the normal as in a regular grid with unit length, i.e. in computational space. Thereafter, the following relationship can be used to convert between physical coordinates p_i and computational coordinates ξ_j, determined using the Jacobian matrix J_{ij}:

$$x_i = J_{ij} \cdot \xi_j \qquad \text{and} \qquad J_{ij} = \frac{\delta p_i}{\delta \xi_j} \tag{6.8}$$

This equation allows the conversion of vectors from one space to another. The missing link is the still unknown Jacobian matrix for a grid node, which is approximated by using central differences:

$$J_{ij} = \frac{1}{2} \cdot \left(\frac{p_i(x_n, y_n, z_n) - p_i(x_{n-1}, y_n, z_n)}{\xi_j(x_n, y_n, z_n) - \xi_j(x_{n+1}, y_n, z_n)} + \frac{p_i(x_{n+1}, y_n, z_n) - p_i(x_n, y_n, z_n)}{\xi_j(x_{n-1}, y_n, z_n) - \xi_j(x_n, y_n, z_n)} \right). \tag{6.9}$$

Since the computational grid was defined with unit length, both denominators are set to one, and hence only the neighboring vectors need to be subtracted. Since central differences must be computed separately for the x, y, and z component of each point p and ξ, the index differs for the component. In the above equation, central differences for the x dimension are computed, i.e. $J_{ij} = J_x$. The vectors for J_y and J_z can be computed similarly. A linear combination of these vectors, i.e. J_x, J_y, J_z, with the normal vector in computational space N^c, results in physical space N^p:

$$N^p = N^c_x \cdot J_x + N^c_y \cdot J_y + N^c_z \cdot J_z \tag{6.10}$$

where N^c_x, N^c_y, and N^c_z denote the x, y, and z components of the normal N^c in computational space, respectively. Since this calculation involves only simple mathematical operations, this can either be performed on the fly, or stored with the volume. For the small demonstration models, the second approach was chosen since the added storage cost is negligible.

6.4.5 Memory Requirements

As pointed out in Section 2.2.3, in addition to that of the scalar value, the explicit storage of each grid vertex requires another 12 bytes (using of 32-bit floating point precision), resulting in a total of 16 bytes per grid node.

Saving a normal vector per grid node adds another 12 bytes, which leads to a total of 28 bytes per vertex grid node. Note that neither adjacency nor a *face-id* information must be stored since this is already implicitly defined in the grid. Hence, in terms of storage consumption it makes no sense to convert a curvilinear grid into a tetrahedral mesh. However, in terms of performance, using the tetrahedral grid traverser might lead to better performance. This issue is addressed in the next section, which discusses the performance results of both approaches in detail, including a comparison of both.

6.4.6 Scalability and Comparison Measurements

Again, all measurements were performed on a shared-memory system with eight dual-core 2 GHz Opteron 870s, and 64 GB RAM. The arithmetic mean from four different viewpoints (three directly derived from an orthogonal viewing positions and one from a perspective position) is presented for the reasons explained in Section 6.3.6. This is reasonable, since an incremental traversal is investigated, where only the primitives traversed are exchanged. This will lead to an interesting comparison in the final results section of the tetrahedral and hexahedral traversal algorithms, as the *Blunt-fin* data set was available as both tetrahedral mesh and hexahedral grid.

In this section, however, the focus is on scalability and a comparison between the pure Plücker traversal and the hybrid traversal. As previously mentioned the hybrid approach using Plücker and bilinear patches delivers better rendering quality compared to the pure Plücker traversal, since the latter approach produces artifacts caused by an additional decomposition of each face. The hybrid algorithm is furthermore able to process more hexahedra per seconds compared to the pure Plücker traversal. This should produce higher frame rates, which is demonstrated in the following.

In this case, only two data sets were available for measurements. The *Blunt-fin* data set is already known as tetrahedral mesh. Here, it is a hexahedral grid represented by 40x32x32 vertices. The *Combustion Chamber*, on the other hand consists of 57x33x25 grid points. Table 6.3 presents the performance of both approaches for direct comparison. The linear scalability with respect to the number of processor cores is again no surprise. More interestingly, the hybrid approach delivers not only better quality, but also better performance than the Plücker traversal when rendering iso-surfaces.

The same conclusions can be drawn for semi-transparent rendering presented in Table 6.7. Now, all values along the ray must be accumulated instead of searching for a specific value. Due to the different termination criteria, more cells along a ray have to be processed which was approximately 20 % for both tested data sets. This is a little less than to the 30% increase

		Number of Cores (fps)									
Data	Scr	1		2		4		8		16	
Set	Res	P	H	P	H	P	H	P	H	P	H
Blunt	512^2	0.97	1.20	1.91	2.45	3.82	4.94	7.57	9.94	12.24	16.24
	1024^2	0.24	0.31	0.48	0.61	0.92	1.23	1.91	2.50	3.74	4.88
Comb	512^2	1.24	1.34	2.47	2.80	4.97	5.67	9.86	11.38	18.85	20.72
	1024^2	0.31	0.34	0.63	0.71	1.25	1.42	2.49	2.86	4.91	5.60

Table 6.6: A direct comparison between both approaches reveals that hybrid traversal (H) delivers not only better rendering quality, but is also significantly faster compared to the pure Plücker traversal (P). This is especially true for small screen resolution (512^2). The difference is not that great for 1024^2, which is probably due to better cache usage.

for the tetrahedral case for the *Blunt-fin* data set, but is probably caused by the fact that the average number of hexahedra is smaller compared to the average number of tetrahedra along a ray. Supposably, each hexahedra of the *Blunt-fin* was decomposed into five tetrahedra to convert the data set (see Figure 6.1).

One unusual finding is that semi-transparent rendering is not always slower than iso-surface rendering. Since these cases are fairly restricted to the *Combustion Chamber*, it is believed that early ray termination pays better off, i.e. the resampled values accumulate rapidly to one. The traversal stops in this case and the accumulated value is returned as pixel color, which leads to higher frame rates.

		Number of Cores (fps)									
Data	Scr	1		2		4		8		16	
Set	Res	P	H	P	H	P	H	P	H	P	H
Blunt	512^2	0.78	1.01	1.53	2.01	3.08	4.09	6.12	8.10	11.29	14.82
	1024^2	0.19	0.25	0.37	0.51	0.76	1.02	1.53	2.03	3.02	3.94
Comb	512^2	1.60	1.77	3.14	3.62	6.29	7.26	12.61	14.45	23.95	24.83
	1024^2	0.41	0.45	0.77	0.88	1.58	1.81	2.92	3.64	6.26	7.08

Table 6.7: The measured frames per second for semi-transparent rendering confirms the conclusion that the hybrid approach (H) is a little faster compared to the pure Plücker traversal (P). Interestingly, the performance for the Combustion *is in some cases even faster compared to iso-surface rendering.*

While the effect of early ray termination needs to be investigated further, this algorithm again achieves a linear scalability. Two dual-core processors allow for interactive frame rates at low resolution (i.e. 512^2), while a high resolution (i.e. 1024^2) demands four times as much. A more important result is that the hybrid hexahedral traversal not only deliver better quality, but is in every case faster than the pure Plücker traversal.

6.5 Results

The system used for performance measurements was equipped as mentioned in Sections 6.3.6 and 6.4.6: a Linux box consisting of eight dual-core 2 GHz Opteron 870s and 64 GByte of shared main memory. Since scalability was already tested in Sections 6.3.6 and 6.4.6 this section compares both approaches against each other on a single dual-core Opteron only. The *Blunt-fin* data set was especially useful since it was available as both tetrahedral mesh and hexahedral grid. All other data sets were only available either as a tetrahedral mesh (*Bucky-ball*, *Orbital*) or as a hexahedral grid (*Combustion Chamber*) only. Table 6.8 shows the key features of each data set.

Data Set	Type	Primitives	Dimensions
Blunt-fin	tetrahedral	224,873	n/a
Orbital	tetrahedral	148,955	n/a
Bucky-ball	tetrahedral	176,856	n/a
Blunt-fin	hexahedral	40,960	40x32x32
Combustion	hexahedral	47,025	57x33x25

Table 6.8: Several unstructured and semi-structured data sets listed with respect to their type, number of primitives, and dimensions. Grid dimensions are not applicable for unstructured data sets.

Putting all parts of the previously discussed incremental traverser together allows for an overall performance comparison. It should be noted that this is in no way comparable to the hierarchical approach for regular grids (see Chapter 4). Since *all* primitives along the ray are visited, basically any volume rendering technique (see Section 2.2.5) can be implemented without any further extension or approximation. Additionally, unstructured grids are far more complicated to handle than regular grids. Hence it is pointless to compare grid traversal for unstructured and structured grids.

It is also true that an incremental traverser exhibits different performance behavior than a hierarchical traverser. To take the viewpoint dependent frame rate into account, four different viewpoints were measured independently. However, only the arithmetic mean was presented in the following performance measurements, despite the fact that the frames per second varies up to 30%.

Three viewpoints were directly derived from the orthogonal viewing axes, i.e. one from the x, from the y, and from the z direction. The fourth viewpoint was chosen from a typical perspective view. It is assumed that most applications demand not a worst-case, but an average case analysis. Understanding these three issues helps to interpret the following table, which summarizes the results of this chapter. Since the scalability of both traverser was already proven in Sections 6.3.6 and 6.4.6, only the performance for one dual-core Opteron 870 is presented here.

		Number of Cores (fps)							
Data	Scr	initial		iso		mip		semi	
Set	Res	1	2	1	2	1	2	1	2
Blunt-fin	512^2	14.23	27.57	0.57	1.13	1.02	2.04	0.81	1.62
	1024^2	3.89	7.00	0.14	0.28	0.27	0.53	0.20	0.40
Bucky-ball	512^2	10.45	19.70	0.42	0.84	0.48	0.98	0.72	1.48
	1024^2	2.78	5.13	0.12	0.24	0.13	0.25	0.19	0.38
Orbital	512^2	10.03	20.19	0.56	1.14	0.56	1.13	0.59	1.21
	1024^2	2.77	5.26	0.27	0.48	0.14	0.28	0.16	0.31

Table 6.9: Measurements in fps for three common unstructured data sets using one or two processor cores. Initial *indicates fps for finding the first primitive along the ray, i.e. by a kd-tree traversal. The columns that follow represent the total rendering time for iso-surface rendering (iso), maximum-intensity-projection (mip), and semi-transparent rendering (semi).*

Table 6.9 shows the performance in frames per second achieved on one and two cores of a dual-core processor for all important volume rendering techniques. *Initial* indicates the kd-tree performance for finding the first primitive along a ray only. This first step consumes 6 to 18% of the total rendering time. The columns that follow represent iso-surface rendering (iso), maximum-intensity-projection (mip), and semi-transparent rendering (semi) including the finding of the initial primitive.

All iso-surfaces are rendered with enhanced normal calculation, leading to about half the speed compared to the straightforward implementation. This is caused by the computationally expensive calculation of the 3D barycentric

coordinates for the intersection point with the tetrahedron. This is necessary to assign meaningful weights for the pre-computed vertex normals. However, the observed performance reduction is less significant for the higher screen resolution. It is believed that in this case, more rays hit the same tetrahedra, thus improving cache usage.

Maximum-intensity projection performs slightly worse in all cases compared to semi-transparent rendering. This seems at first implausible since the ray integration is replaced by a simple comparison. However, semi-transparent rendering can obviously profit from early ray termination, i.e. stopping traversal if the accumulated intensity reaches one.

		Number of Cores (fps)							
Data	Scr	initial		iso		mip		semi	
Set	Res	1	2	1	2	1	2	1	2
Blunt-fin	512^2	14.23	27.57	1.20	2.45	1.23	2.44	1.01	2.01
	1024^2	3.89	7.00	0.31	0.61	0.31	0.61	0.26	0.51
Comb	512^2	9.36	18.31	1.34	2.80	0.70	1.39	1.77	3.62
	1024^2	2.60	4.70	0.34	0.71	0.17	0.33	0.45	0.88

Table 6.10: Measurements in fps for two common curvilinear data sets using one or two processor cores. Initial *indicates fps for finding the first element along the ray. The columns that follow state the total rendering time for iso-surface rendering (iso), maximum-intensity-projection (mip), and semi-transparent rendering (semi).*

Section 6.4.6 already showed that the proposed hybrid algorithm, i.e. using *Plücker coordinates* and bilinear patches, produces not only better quality but is also significantly faster. It is no surprise that this is also true for maximum-intensity projection, where a performance gain of 10 - 50% compared to the pure *Plücker* traversal can be expected. Table 6.10 therefore shows only fps for the hybrid approach.

Of course, a variation between 10 - 50% in performance is rather high, but actually true for other measurements too. The reason for this can be found in the bilinear patch intersection test. As already noted, this is basically solving a quadratic equation. The square root coefficient can, however, get very small leading to numerical instabilities. If this occurs, a linear equation is solved. Computing this linear solution is far less expensive since there is no square root involved, leading to a faster traversal speed. This is true for large parts of the *Blunt-fin* data set, i.e. from a certain viewpoint, the visible parts of the *Blunt-fin* set mostly contains patches where numerical issues disable

the quadratic solution. The Plücker space traversal cannot of course profit from this special case.

Another strange result seems to be the fact that the *Blunt-fin* is rendered more slowly than the *Combustion Chamber* using semi-transparent rendering, while this behavior is reversed for maximum intensity projection. It can be concluded that on average the accumulation evaluates to one even before the iso-surface is found. Maximum-intensity projection cannot profit from this behavior. Additionally, the data set contains hardly quadrilaterals, making the bilinear patch evaluation computationally expensive.

6.6 Conclusion

This chapter provided a thorough discussion of the implementation of an incremental traversal algorithm for unstructured and semi-structured volume data sets. The main idea relies on converting all edges of a primitive into so-called *Plücker coordinates*, allowing for a quick decision whether a ray passes a primitive edge clockwise or counterclockwise. The efficiency of this approach results mainly from the property that many edges, except for the boundary faces, are shared between two adjacent primitives and can therefore be re-used when selecting the next primitive along a ray. Incremental traversal was designed such that the number of *Plücker tests* is stable for each primitive, so that the rendering speed relies only on the number of primitives traversed.

The tetrahedral traversal code requires 2.67 tests per tetrahedron, which is for reasons explained, faster than the theoretical optimum of 2.33 tests. This observation depends on the processor architecture and is therefore subject to change for other platforms or future generations of processors. The storage cost, with an additional 16 bytes per tetrahedron, is quite tolerable, since other approaches reported up to 160 bytes per tetrahedron [Weiler03]. Furthermore, it is worth noting that only one vertex needs to be fetched per newly traversed tetrahedron, keeping memory bandwidth low. Additionally, no *Plücker coordinates* are pre-computed. Although this would lead to some speedup compared to converting the ray and all edges along the ray during traversal, it requires additional storage of six floating-point values per edge. Furthermore, it must be kept in mind that *Plücker coordinates* represent *oriented lines* in space, and hence correct orientation must be computed before usage. Implementing pre-computed *Plücker coordinates*, therefore, would probably introduces additional conditional statements that destroy the advantages of modern pipelined processors.

The hexahedral traversal was first implemented using the same ideas as for the tetrahedral case. Since hexahedra are more complex, the number

of tests increase to four tests per hexahedron. This is only true for convex tetrahedra. Handling concave hexahedral faces is possible, but more complicated. The hybrid approach uses a combination of *Plücker tests* and bilinear patch intersections, and needs therefore only two *Plücker tests* per hexahedron, from which at most one could be used for the next hexahedron along the ray. However, multiple conditional statements when reusing such results is an issue, making the repeated computation a faster alternative.

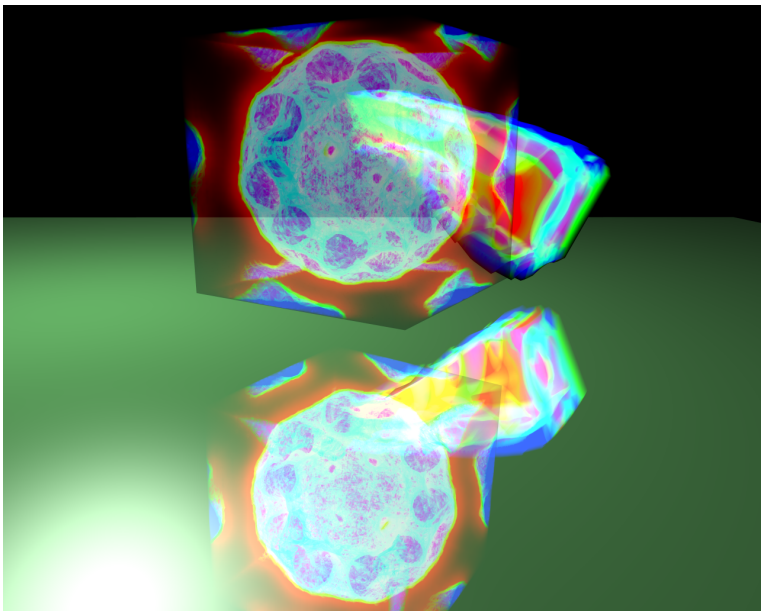


Figure 6.16: Unstructured (Bucky-ball) *and curvilinear* (Combustion Chamber) *data sets not only can be rendered into one scene, their primitives can also interact with each other.*

The main advantage of an incremental traversal algorithm is, of course, that all volume rendering techniques can be easily implemented. An additional advantage of Plücker space traversal is its usage for non-planar faces. For example, ray-plane intersection can be used for tetrahedral meshes, but not for curvilinear grids, since hexahedral faces generally do not consist of planar faces. In fact, *Plücker coordinates* can be used for many other primitives, like prisms, etc. The only restriction is that every connection between two vertices can be represented as a straight line. On the other hand, the methods introduced by [Garrity90] and [Frühauf94] are restricted to a certain primitive, i.e. tetrahedra or hexahedra and therefore less versatile.

Another effect results from integrating this technique into a larger ray tracing system. This makes it possible to combine different primitives easily

into one scene. Basically, the arguments discussed in Section 4.3.5 hold here as well. This works without any additional effort, i.e. once a shader is available for one primitive, it can be used by any primitive located in the scene. A sample demonstration of this scene integration is shown in Figure 6.16.

This versatility, of course, comes with a price. Interactive frame rates requires clustering of several processors, especially for larger screen resolutions. In this case, 8 or even 16 cores are needed to provide interactivity. On the other hand, quad-core processors are already available, and the CELL architecture even offers eight parallel units. In the near future it can therefore be expected that even more cores will be packed on one processor allowing for interactive frame rates on a consumer PC. Additional interesting areas of investigation are discussed in the last section of this chapter.

6.7 Contributions

The author's contributions to the topics discussed in this chapter are:

1. The author completely developed and implemented the incremental tetrahedral traversal algorithm in [Marmitt06c] including iso-surface rendering, maximum-intensity-projection, and semi-transparent rendering. This also includes the integration into the OpenRT framework and all presented performance measurements.

2. The author completely developed in implemented the incremental hexahedral in [Marmitt06c] traversal algorithm including the hybrid approach based on bilinear patches. Like before, the hexahedral traversal was integrated into the OpenRT framework and the performance was measured by the author, too.

6.8 Future Work

This chapter initially discussed the concept of *Plücker coordinates* for traversing tetrahedral meshes. Unstructured data sets are far more complicated to handle, so that in contrast to Chapter 4 and 5, an incremental traverser was implemented allowing for a variety of volume rendering techniques (see Section 2.2.5) with little effort. Although it is best suited for semi-transparent rendering, iso-surface rendering, and maximum-intensity projection are also possible to implement.

However, the last two rendering techniques would work better in conjunction with a hierarchical acceleration structure. Parker et al. [Parker99b]

sorted the entire tetrahedral mesh into a grid for quick identification of the cells containing the iso-surface. This combination is applicable here, too. Multi-level grids have the advantage of low memory consumption for the hierarchy, but lack flexibility. Two other major spatial acceleration structures, the kd-tree and the bounding volume hierarchy [Wald07], may be better choices today since today memory consumption is a minor issue. Wald et al. [Wald07] report frame rates 30 to 40 times higher compared to the incremental traverser here using a comparable viewport and data set. Octrees are also used today [Knoll06], but seem better suited for regular grids.

Basically the same is true for maximum intensity projection. Again Parker et al. [Parker99b] showed a rather simple method of acceleration by implementing a priority queue, which skipped regions within the volume where the maximum is below the maximum currently found.

Semi-transparent rendering, especially when using transfer functions, can also be sped up further. The key observation here is that pre-integrated volume rendering [Engel01] can also be used for tetrahedral meshes due to their linear behavior. This was already demonstrated by Weiler et al. [Weiler03] in their graphics card volume renderer. Currently, this result is achieved by simply resampling more points along the ray, which, without an adaptive scheme, is time-consuming.

The same ideas can be applied to curvilinear data with respect to visualization methods. Here also, iso-surface rendering and maximum-intensity projection would certainly benefit from using a hierarchical acceleration structure. If linear approximation suffices, pre-integrated volume rendering is option here too. On the other hand, a significant speedup can be expected when optimizing the fetching of new vertices while traversing the grid. A nice feature of the tetrahedral traverser was that only one vertex per new tetrahedron must be fetched. Curvilinear grids consist of hexahedra, which requires four vertices be loaded. Analysis shows that this reduces the number of processed hexahedra per second by 25%. Bricking or the prefetching mechanism might lead here to further improvements.

Both algorithms can be further accelerated by shooting rays in parallel, i.e. using SIMD operations. This is, however, difficult for an incremental traverser since the rays diverge quickly for perspective viewing. In combination with an acceleration structure, however, the same principles could be used as already discussed in Chapter 4 and recently demonstrated in [Wald07]. Further ideas for optimizing the traversal can be found in [Reshetov05], as well as for the CELL architecture in [Benthin06]. Optimal cache usages are an issue for all modern processors and therefore require special attention.

As already discussed in Section 4.8, a combination of a hierarchical and incremental approach seems to be the optimal approach for a versatile volume

rendering tool. This leads directly to the last chapter of this thesis, which summarizes the major topics.

Chapter 7
Final Summary

Im Auslegen seit frisch und munter, und legt Ihr's nicht aus, so legt was unter!

Johann Wolfgang von Goethe

This thesis is a contribution towards a flexible and interactive volume ray tracing system for scientific visualization. Chapter 2 showed that ray tracing allows for an appropriate and straightforward implementation of iso-surface rendering, maximum-intensity projection, and semi-transparent rendering. This is true for all major volumetric types, i.e. regular and curvilinear grids, or unstructured meshes. There was never a doubt concerning the quality of the images rendered with ray tracing. Rendering performance, on the other hand, was always considered as too slow. Since scientific visualization tasks aim at exploring new features, interactive walk-through and adjustment of visualization parameters, e.g. iso-value or control points of transfer functions are a 'must-have' feature of every renderer.

Hence, much research was devoted to alternative approaches more suitable for GPU implementations. Most implementations aim at adding different visualization methods to modern graphics adapters. Such graphic adapters are usually designed to render only triangles efficiently and therefore does not take interaction with a media into account. Cell projection, vertex projection (splatting), and texture mapping allow rendering of volumetric data. Using hardware for rendering, even if it was not specifically designed for it, is of course quite fast but suffers from serious limitations. Cell projection requires a visibility sorting for every viewpoint change; vertex projection lacks quality, and texture mapping is restricted to regular grids and to the size of texture memory. Therefore, several hardware architectures were developed which resulted in an interactive high-quality volume rendering system. It soon turned out that such custom hardware was outdated before it was ready for market. Today, only the *VolumePro* board is sold by TerraRecon. Other developments, like *Cube*, *Vizard*, and *Virix*, have faded away in the meantime.

Instead of trying to increase the quality of GPU-approaches, the idea here was to make ray tracing fast enough for interactive purposes. This has been pursued by numerous researchers, especially since graphic adapters are now programmable and allow, e.g. the implementation of a volume ray caster using a fragment shader. While this is a possible area of research, it is believed that ray tracing itself should first be further developed in software before adapting it to any kind of hardware. Software ray tracers have an additional advantage of seamlessly integrating different primitives into one scene (see Figure 7.1), which is still difficult to achieve even on modern graphic adapters.

To this end, two basic ideas were extensively developed throughout Chapters 4 - 6. Chapter 4 introduced the concept of the *implicit kd-tree* for iso-

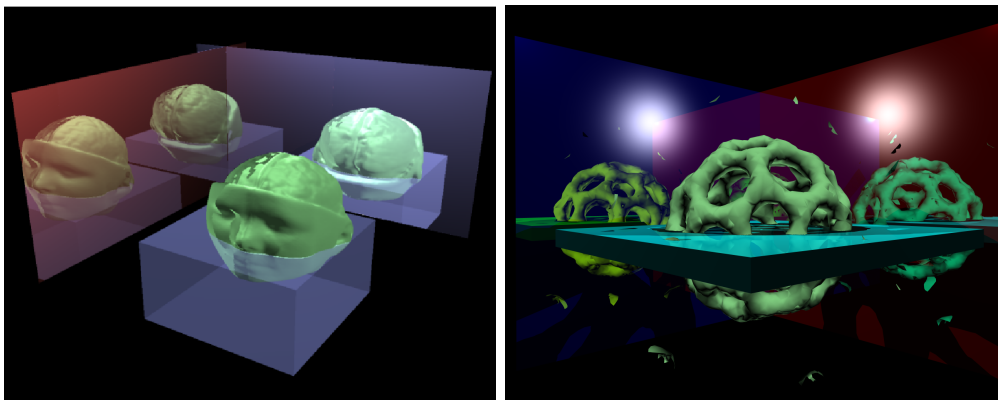


Figure 7.1: Left: *A test scene showing mixed surfaces and interactive iso-surface volume rendering with roughly 2 fps at* 640×480 *resolution on a dual Intel Pentium-4 2.2 GHz system.* Right: *Seamless integration with surface ray tracing. The volume data set, in this case the Bucky-ball, is augmented and surrounded by reflective surfaces and light sources.*

surface rendering of regular grids. With this hierarchical data structure, it is possible to render even large volume data sets on a single PC. The key observation was the fact that only a small subset of all cells actually contains the iso-value sought at any given time. Since iso-surface rendering is basically a range searching problem, adapting the kd-tree to allow for searching cells containing the iso-surface was the natural next step. The kd-tree nodes were augmented with minimum and maximum values representing the range of the associated sub-volume within the grid. This allows for an efficient culling of subtrees but consumed too much memory in its naïve implementation. An optimized version stored only the two range values assuming the tree was balanced. Newly introduced implications, i.e. the power-of-two constraint, are relaxed. This reduces memory consumption already from 16 times to only twice as much memory as the original data set. Furthermore, simple quantization schemes showed that additional memory can be saved without reducing the performance too much.

In the second step, the actual intersection with the iso-surface within a located cell must be computed to finally render an iso-surface. Until recently, it was common to derive a cubic polynomial, which was then solved with the Cardanos formula. Unfortunately this introduce trigonometric functions, which are costly to compute. Therefore it was proposed to replace the Cardanos formula with an iterative root finding scheme. This scheme divides the cubic polynomials into three parts, depending on the computed extrema

values. Since there is exactly zero or one ray intersection per segment, only the first segment where the range lies between the iso-values, must be investigated further. Iterative bi-section determines the exact intersect position by using a small number of loops. This new approach is as fast as Neubauers iterative linear interpolation, but does not suffer from the problem of returning no intersection if two or more intersections exist along a ray.

For the kd-tree, as well as the computation of the intersect position within a cell, both a single ray and a packet ray variant were implemented. Using SIMD compiler intrinsics, it is possible to traverse the kd-tree with four rays in parallel. This led to a speedup factor between two and the theoretical maximum of four. Only in scenarios where a small cell size is combined with low resolution, is the speedup close but never below to single ray performance.

Although this system is able to handle even large data sets, loading and traversal speed could be improved. An LOD scheme together with the *treelet* concept and an improved MMU, not only reduces the time for loading the data set to one third, but also doubles the traversing speed and saves memory.

The next obvious step was to extend this framework to allow for semi-transparent rendering. To make use of the hierarchy, the kd-tree nodes are further augmented with an arithmetic mean, which is used to skip homogeneous regions. Visual results, as well as statistical error metrics, are promising, which makes using this acceleration structure for semi-transparent rendering worthwhile. However, at the current state, neither transfer functions nor high-quality is efficiently supported. Suggestions addressing these two issues were made in Section 4.8.

Chapter 5 continued extending the *implicit kd-tree*. This time the temporal domain was addressed, allowing the rendering of time-dependent regular grids. Two approaches were discussed to extend the existing rendering framework. In the first approach, the kd-tree was updated completely for every time step concurrently. This was achieved by carefully optimizing the existing tree building procedure in two ways. Most importantly, the recursive build was replaced by an iterative update, which worked in conjunction with bricking and other minor optimizations at a rate three times faster than a straightforward recursive implementation. Secondly, this allowed the kd-tree to be updated in parallel by employing several processors. Finally the original client-server-based implementation needed to be adapted for the shared-memory system used here. This was necessary since not every render client may build its own tree in each step, but rather may divide the system into render and update nodes carefully chosen with respect to the size of the updated volume.

Although this approach allows for streaming applications, i.e. it is independent in the number of time steps and does not need an extensive prepro-

cessing step, the dimensional size of a single time step is certainly a limiting factor. It was therefore proposed to update the tree not for a single, but only after some (not necessarily fixed) number of time steps, thus relaxing the burden of real-time update. For this purpose, the 4D kd-tree could be used as a second method for rendering time-dependent volumetric data.

This tree is simply an *implicit kd-tree* augmented with a temporal dimension. Its key feature is that there is no separation between temporal and spatial domains, which is the usual way for handling time-dependent data [Shen99, Younesy05]. In its current implementation, this structure is unfortunately not competitive since it consumes as much memory as the concatenation of all individual trees and runs 30% slower. The latter issue was addressed by a more clever ordering of the splitting dimension, but the memory consumption is still too high. In all, it seems to be a promising structure for future research, especially since the 4D kd-tree also works for larger data sets. Both approaches work and are complementary. Small data sets can be rendered with a concurrent kd-tree update without worrying about the number of time steps. Larger data sets require pre-processing and should therefore use the 4D kd-tree.

The next sections of Chapter 5 showed the versatility of the *implicit kd-tree* for other applications. As a first example, a multi-level instancing approach for rendering large terrains was briefly described, along with how the *implicit kd-tree* identified the surface patches pierced by a ray. This required only minor modifications since the elevation values could be treated as range (iso-) values. The entire system allows the rendering of large terrains populated with thousands of plants, consisting of up to 90 trillion triangles at near-interactive frame-rates.

The second example showed a concept very similar to the concurrent tree update supporting dynamic rendering of polygonal data. A *bounded kd-tree* combines the strengths of ordinary kd-trees and bounding volume hierarchies into a single acceleration structure. Both splitting plane positions and orientations attached to each node can be interpreted as a certain range which is very similar to the ranges of iso-values stored in an *implicit kd-tree*. If the temporal distribution of scene triangles is not random, interactive frame-rates are certainly possible.

While Chapters 4 and 5 described and extended the *implicit kd-tree* as a hierarchical data structure, step by step to finally allow semi-transparent rendering, Chapter 6 started with a semi-transparent renderer. Here, a new versatile incremental traverser was introduced that offers direct and high-quality support of different rendering tasks, i.e. iso-surface rendering, maximum-intensity projection, and semi-transparent rendering. It is also adaptable to any underlying convex primitive as long as its edges can be rep-

resented as straight lines. This was successfully demonstrated for tetrahedra and hexahedra primitives.

Plücker coordinates are the key for fast traversal of both tetrahedral meshes and hexahedral grids. Traversing tetrahedral meshes is possible with just 2.67 *Plücker tests* and one vertex fetch per tetrahedron. When using small screen resolutions, a dual-core processor suffices for interactive frame rates. Larger resolutions and larger data sets require either a shared-memory system or a cluster of PCs.

Handling hexahedra primitives complicates the traversal process, since the number of faces increases and each face consists of four vertices instead of three. Assuming convex hexahedra, four *Plücker tests* suffice to determine the exit face, plus one for calculating an interpolation point. It turns out that this leads to rendering artifacts caused by the newly introduced diagonal on a face. Therefore a hybrid approach was suggested that is not only faster compared to a pure Plücker traversal, but avoids artifacts by using bilinear patches for the parameterization of the intersected hexahedral faces. Two *Plücker tests* first sort out two of the five possible exit faces. The three remaining faces are treated as bilinear patches, where parametric coordinates ranging between zero and one indicate the patch intersection with a ray, and hence the exit face. This must be applied at most three times, which is important due to the computationally expensive square root operation inherent in the bilinear patch computation.

Furthermore, the *Blunt-fin* model showed that it makes no sense to convert a curvilinear grid into an unstructured mesh by decomposing each hexahedra into five tetrahedra. Both storage cost and rendering time increase compared to the hexahedral traverser. However, incremental traversal algorithms are perfectly suited for semi-transparent rendering but not as useful for iso-surface rendering and maximum-intensity projection in terms of performance. For such tasks it is better to add a spatial hierarchy.

In other words, Chapters 4, 5, and Chapter 6 approached the interactive volume ray tracing problem from two different perspectives. Chapter 4 and 5 introduced a hierarchical structure, which was later extended allowing semi-transparent rendering. It turns out that this can only be efficiently solved by switching to a grid traverser if a non-homogeneous region is encountered during the hierarchical traversal. Chapter 6 described, on the other hand an incremental traverser that must be extended with a spatial hierarchy for efficient iso-surface rendering. Augmenting each approach with its missing part would therefore lead to a flexible and interactive volume ray tracing system specified in the introductory chapter.

Zusammenfassung

Diese Arbeit leistet einen Beitrag zu einem flexiblen und interaktiven Volumen-Ray-Tracing-Systems für die wissenschaftliche Visualisierung. Kapitel 2 zeigte, dass Ray-Tracing eine geeignete und einfache Implementierung von Isoflächen-Darstellung, Maximalintensitätsprojektion und semi-transparenter Darstellung ermöglicht. Dies trifft auf die wichtigsten volumetrischen Topologien zu, i.e. reguläre und gekrümmte Gitter oder unstrukturierte Netze.

Obwohl es grundsätzlich keinen Zweifel bezüglich der Darstellungsqualität gibt, wurde die Bildgenerierung immer als zu langsam angesehen. Da die wissenschaftliche Visualisierung darauf abzielt, neue Eigenschaften zu entdecken sowie interaktives Positionieren der Kamera und die Änderung anderer Parameter, beispielsweise Stützpunkte von Transferfunktionen zu ermöglichen, sind solche Funktionalitäten für bildgebende Verfahren ein Muss.

Deshalb wurde einiges an Forschung in alternative Ansätze eingebracht, welche GPU-orientert sind. Die meisten Implementierungen zielen darauf ab, verschiedene Visualisierungsmethoden modernen Grafikkarten hinzu zufügen. Solche Grafikkarten sind typischerweise entworfen worden, um Dreiecke effizient darzustellen, was wiederum gerade das Gegenteil eines semi-transparenten Mediums ist. Zellprojektion, Knotenprojektion und Textureabbildungen erlauben das Darstellen volumetrischer Daten. Die Benutzung von Hardware zur Darstellung, selbst wenn diese nicht dazu entworfen worden ist, ist natürlich relativ schnell, aber bringt auch nicht unerhebliche Beschränkungen mit sich. Zellprojektion benötigt eine Sichtbarkeitssortierung, sobald sich der Beobachtungspunkt ändert; Knotenprojektion erreicht nicht die erwünschte Qualität und Textureabbildungen sind nur für reguläre Gitter anwendbar und zudem limitiert durch die Größe des Texturespeichers. Deshalb wurden verschiedene Hardware-Architekturen entwickelt, die zu einem interaktiven System mit hoher Ausgabequalität führten. Es stellte sich jedoch bald heraus, das solche spezial-gefertigte Hardware veraltet war, bevor die Marktreife erreicht wurde. Heute wird nur noch die *VolumePro*-Karte von TerraRecon verkauft. Andere Entwicklungen, wie *Cube*, *Vizard* und *Virix* sind in der Zwischenzeit eingestellt worden.

Statt die Qualität von GPU-Ansätzen zu verbessern, wurde hier die Idee verfolgt, Ray-Tracing für interaktive Zwecke ausreichend zu beschleunigen. Dies wurde schon von einigen Forschern betrieben, insbesondere seit Grafikkarten nun programmierbar sind und so beispielsweise die Implementierung eines Volumen-Ray-Casters im Fragment-Shader erlauben. Obgleich dies ein möglicher Forschungsansatz ist, wurde es hier vorgezogen, Ray-Tracing selbst zuerst in Software weiter zu entwickeln, bevor man es auf irgendeine Hardware adaptiert. Software-Ray-Tracer haben den zusätzlichen Vorteil, dass verschiedene Primitive in der Szene nahtlos integriert werden können (siehe Bild 7.2), was selbst auf modernen Grafikkarten immer noch schwierig zu realisieren ist.

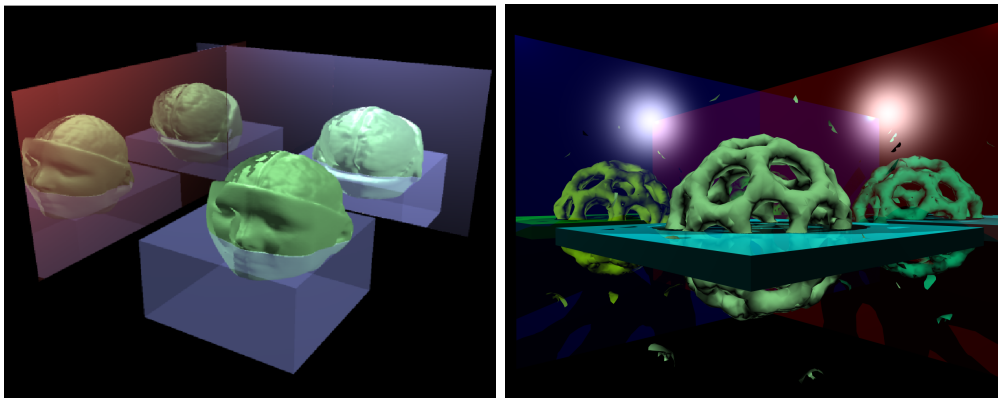


Abbildung 7.2: Links: *Eine Test-Szene zeigt verschiedene Oberflächen zusammen mit einer Isofläche interaktiv mit ungefähr zwei Bildern pro Sekunde bei einer Auflösung von* 640×480 *auf einem Dual Intel Pentium-4 2.2 GHz-System.* Rechts: *Nahtlose Integration mit Oberflächen-Ray-Tracing. Der Volumendatensatz, in diesem Fall ein Buckminster Fulleren, ist von spiegelnden Oberflächen und Lichtquellen umgeben.*

Um dies zu erreichen, wurden zwei grundlegende Ideen in den Kapiteln 4 - 6 detailliert entwickelt. Kapitel 4 führte das Konzept des *impliziten Kd-Baums* für Isoflächen-Darstellung von regulären Gittern ein. Mit dieser hierarchischen Datenstruktur ist es möglich, auch große Datensätze auf einem einzelnen Rechner darzustellen. Schlüssel dazu war die Erkenntnis, dass nur ein kleiner Teil aller Zellen tatsächlich den gesuchten Isowert enthält. Da Isoflächen-Darstellung im Prinzip ein Problem der Bereichssuche ist, war die Anpassung eines Kd-Baumes zur Suche nach Zellen mit der Isofläche der logische nächste Schritt. Die Kd-Baum-Knoten wurden mit Minimum- und Maximumwerten erweitert, die den Bereich des korrespondierenden Teil-Volumens

innerhalb des Gitters speichern. Dies erlaubt das effiziente Überspringen von Teilbäumen, aber verbraucht bei einer simplen Implementierung zu viel Speicher. Eine optimierte Version speichert nur noch die Bereichswerte unter der Annahme, dass der Baum ausgeglichen ist. Die dadurch hinzu kommenden Einschränkungen, wie die Beschränkung auf Zweier-Potenzen, wurden anschließend wieder aufgelöst. Der zusätzliche Speicherverbrauch wurde dadurch bereits von der 16fachen hin zur doppelten Menge im Verhältnis zum Originaldatensatz reduziert. Einfache Quantisierungsschemata zeigten, dass der Speicherverbrauch weiter verringert werden kann, ohne dass die Systemleistung darunter signifikant leidet.

In einem zweiten Schritt muss der eigentliche Schnittpunkt mit der Isofläche innerhalb einer gefundenen Zelle berechnet werden, um die Isofläche letztendlich darzustellen. Bis vor Kurzem war es üblich, ein kubisches Polynom abzuleiten, welches dann mit den Cardano-Formeln gelöst wurde. Unglücklicherweise beinhalten diese trigonometrische Funktionen, deren Berechnung teuer ist. Deshalb wurde vorgeschlagen, die Cardano-Formeln durch iteratives Finden der Nullstellen zu ersetzen. Nach diesem Schema wird das kubische Polynom in drei Abschnitte eingeteilt, die von den Extremwerten abhängen. Da es in jedem Abschnitt nur genau eine oder gar keinen Strahlschnitt gibt, muss nur der erste Abschnitt genauer untersucht werden, in dem der Isowert im jeweiligen Bereich liegt. Eine iterative Halbierung bestimmt den exakten Schnittpunkt dann mit einer geringen Anzahl von Schleifendurchläufen. Diese neue Methode ist ebenso so schnell wie Neubauer's iterative lineare Interpolation, hat aber nicht das Problem, keinen Schnittpunkt zu liefern, wenn zwei oder mehr Schnittpunkte entlang des Strahls existieren.

Sowohl für den Kd-Baum, also auch für die Berechnung des Schnittpunktes innerhalb einer Zelle wurden eine Einzel- sowie eine Paketstrahlvariante implementiert. Durch die Benutzung von übersetzerabhängigen SIMD-Befehlen ist es möglich, den Kd-Baum mit vier Strahlen gleichzeitig zu durchlaufen. Dies führte zu einem Beschleunigungsfaktor zwischen zwei und dem theoretischen Maximum von vier. In den Szenarien, in denen eine Kombination von kleinen Zellgrösen und geringer Auflösung vorlag, ist die Beschleunigung zwar nur noch marginal größer, aber nie langsamer als die Einzelstrahlversion.

Obwohl dieses System bereits in der Lage ist, auch große Datensätze zu handhaben, konnten das Laden und das Durchlaufen des Kd-Baumes weiter verbessert werden. Eine Hierarchie unterschiedlicher Detailierungsstufen zusammen mit dem *Treelet*-Konzept sowie einer verbesserten Speicherverwaltung reduzierte nicht nur die Zeit zum Laden des Datensatzes auf ein Drittel, sondern verdoppelte auch die Durchlaufgeschwindigkeit des Kd-Baumes und verbraucht weniger Speicher.

Der nächste offensichtliche Schritt war nun, dieses System um eine semitransparente Darstellung zu ergänzen. Um die Hierarchie des Kd-Baums auszunutzen, wurden die Knoten um ein arithmetisches Mittel erweitert, dass dazu benutzt werden kann, homogene Regionen zu überspringen. Die visuellen Ergebnisse sowie statistische Fehlerabschätzungen sind vielversprechend, wodurch die Benutzung dieser Beschleunigungsstruktur lohnenswert erscheint. Trotzdem, im Moment werden weder Transferfunktionen noch hochwertige Visualisierungen effizient unterstützt. Lösungsvorschläge zu diesen beiden Punkten wurden im Abschnitt 4.8 gemacht.

Im Kapitel 5 wurde der Ausbau des *impliziten Kd-Baums* fortgesetzt. Diesmal wurde die zeitliche Dimension adressiert. Zwei Ansätze wurden besprochen, um das existierende System nachzurüsten. Im ersten Ansatz wird der Kd-Baum für jeden Zeitschritt nebenläufig komplett aktualisiert. Dies wurde erreicht, indem die existierende Prozedur für den Baumbau in zweifacher Hinsicht erweitert wurde. Am Wichtigsten war die Ersetzung des rekursiven Bauens durch ein iteratives Aktualisieren, dass zusammen mit Bricking und anderen kleineren Verbesserung dreimal schneller ist als eine einfache rekursive Implementierung. Zweitens, erlaubte dies gleichzeitig das simultane Aktualisieren eines Baumes durch Benutzung mehrerer Prozessoren. Schließlich musste auch die Client-Server-basierte Implementierung für das hier benutzte Shared-Memory-System angepasst werden. Dies war notwendig, da nicht jeder Darstellungs-Client seinen eigenen Baum für jeden Schritt bauen soll, sondern das System vielmehr in Darstellungs- und Aktualisierungsknoten unterteilt wurde, deren Verhältnis wiederum sorgfältig hinsichtlich der Größe des Volumens gewählt wurde.

Obwohl dieser Ansatz Streaming-Anwendungen erlaubt, er also unabhängig von der Anzahl der Zeitschritte ist und keine umfangreiche Vorverarbeitung benötigt, ist die Größe eines einzelnen Zeitschritts zweifellos ein limitierender Faktor. Deshalb wurde vorgeschlagen, den Baum nicht für einen einzigen, sondern für einige (nicht unbedingt festgelegte) Anzahl von Zeitschritten zu bauen, was wiederum die Belastung durch die Echtzeit-Aktualisierung verringern würde. Für dieses Zweck könnte der 4D Kd-Baum als zweite Methode benutzt werden, um zeitabhängige Volumendaten darzustellen.

Dieser Baum ist einfach ein *impliziter Kd-Baum*; erweitert um eine zeitliche Dimension. Sein Hauptmerkmal ist die Tatsache, dass keine Trennung zwischen zeitlichen und örtlichen Dimensionen statt findet, was sonst für die Handhabung zeitabhängiger Daten üblich ist [Shen99, Younesy05]. In ihrer gegenwärtigen Implementierung ist diese Struktur leider nicht konkurrenzfähig, da sie gleich viel Speicher gegenüber einer Verknüpfung aller einzelnen Bäume verbraucht und sie auch 30 % langsamer ist. Letzteres wurde durch eine intelligentere Anordnung der Schnittebenen beseitigt, aber der

Speicherbrauch bleibt dennoch hoch. Insgesamt erscheint sie jedoch als eine vielversprechende Struktur für weitere Forschungen, insbesondere da der 4D Kd-Baum auch bei großen Datensätzen arbeitet. Beide Ansätze funktionieren und ergänzen sich somit. Kleinere Datensätze können mit der nebenläufigen Kd-Baum-Aktualisierung dargestellt werden, ohne sich über die Anzahl der Zeitschritte Gedanken machen zu müssen. Größere Datensätze erfordern eine Vorverarbeitung und sollten deshalb zur Volumenvisualisierung den 4D Kd-Baum benutzen.

Die folgenden Abschnitte von Kapitel 5 zeigte die Vielseitigkeit des *impliziten Kd-Baumes* in anderen Anwendungen. Als erstes Beispiel wurde ein mehrstufiger Instanziierungsansatz zur Darstellung weiträumiger Landschaften kurz beschrieben, indem der *implizite Kd-Baum* die vom Strahl getroffene Geländekacheln identifiziert. Die Modifizierungen dazu waren nur gering, da die Höhenwerte einfach als Bereichs- (Iso-)werte interpretiert wurden. Das Gesamtsystem erlaubt die Darstellung weiträumiger Landschaften, in denen sich zigtausend Pflanzen befinden und aus bis zu 90 Milliarden Dreiecken bestehen, nahezu interaktiv.

Das zweite Beispiel zeigte ein Konzept, dass der nebenläufigen Kd-Baum-Aktualisierung ähnlich ist und die dynamische Darstellung polygonaler Szenen ermöglicht. Der *einbegrenzte Kd-Baum* kombiniert die Stärken von gewöhnlichen Kd-Bäumen und Boxhierarchien in einer einzigen Beschleunigungsstruktur. Beide Schnittebenenpositionen und -orientierungen werden an jeden Knoten angehängt und können als ein bestimmter Bereich interpretiert werden, der den Bereichsisowerten des *impliziten Kd-Baums* sehr ähnlich ist. Solange die zeitliche Verteilung der Szenendreiecke nicht zufällig ist, sind interaktive Bildwiederholraten fast immer möglich.

Während die Kapitel 4 und 5 den *impliziten Kd-Baum* als eine hierarchische Datenstruktur beschrieben und Schritt für Schritt zur semi-transparenten Darstellung erweiterten, begann Kapitel 6 mit einer semi-transparenten Darstellung. Hier wurde ein neuer, vielseitiger inkrementieller Traversierer eingeführt, der eine direkte und hochqualitative Darstellung verschiedener Darstellungsmöglichkeiten anbietet, insbesondere Isoflächen-Darstellung, Maximalintensitätsprojektion und semi-transparente Darstellung. Er ist auch anpassbar hinsichtlich des darunter liegenden konvexen Primitives, solange dessen Kanten durch Strecken represäsentiert werden können. Dies wurde erfolgreich für Tetraeder und Hexaeder gezeigt.

Plücker-Koordinaten sind der Schlüssel sowohl für eine schnelle Traversierung von Tetraedernetzen als auch Hexaedergittern. Die Traversierung von Tetraedernetzen benötigt gerade einmal 2.67 *Plücker-Tests* und das Nachladen eines Eckpunktes pro Tetraeder. Bei kleineren Bildauflösungen reicht bereits ein Doppelkernprozessor für interaktive Bildwiederholraten. Größe-

re Bildauflösungen und größere Datensätze benötigen dagegen ein Shared-Memory-System oder einen Cluster von PCs.

Die Handhabung von Hexaedern komplizierte den Traversierungsprozess, da sich die Anzahl der Flächen erhöht und jede Fläche nun durch vier statt drei Eckpunkten begrenzt wird. Nimmt man an, dass alle Hexaeder konvex sind, reichen vier *Plücker Tests* um die Austrittsfläche zu bestimmen; sowie ein weiterer Test, um den Interpolationspunkt zu berechnen. Es zeigt sich, dass dies zu Darstellungartefakten aufgrund der neu hinzu gefügten Diagonale führt. Deshalb wurde ein hybrides Verfahren vorgeschlagen, dass nicht nur schneller gegenüber einer reinen *Plücker-Traversierung* ist, sondern auch Artefakte aufgrund der Benutzung Bilinearer Flächen für die Parametrisierung der geschnittenen Hexaederflächen vermeidet. Zwei *Plücker Tests* sortieren dazu zunächst zwei der fünf möglichen Austrittsflächen aus. Die drei verbleibenden Flächen werden als Bilineare Flächen betrachtet, wobei parametrische Koordinaten zwischen null und eins einen Schnitt des Strahls mit der Fläche und damit auch der Austrittsfläche anzeigen. Dies muss höchstens dreimal durchgeführt werden, was aufgrund der immanenten rechenintensiven Quadratwurzel bei der Berechnung Bilinearer Flächen entscheidend ist.

Zudem zeigte das *Blunt-fin*-Modell, dass es keinen Sinn macht, ein gekrümmtes Gitter in ein unstrukturiertes Netz durch Zerlegung jedes Hexaeders in fünf Tetraeder umzuwandeln. Sowohl der Speicherbedarf als auch die Darstellungszeit erhöht sich gegenüber dem Hexaeder-Traversierer. Trotzdem sind inkrementielle Traversierer zwar sehr gut zur semi-transparenten Darstellung geeignet, aber in Bezug auf Leistung nicht unbedingt sinnvoll für Isoflächen-Darstellung und Maximalintensitätsprojektion. Für solche Anwendungen ist es sinnvoller, eine räumliche Hierarchie hinzu zufügen.

Mit anderen Worten, die Kapitel 4, 5 und 6 näherten sich dem Problem der interaktiven Darstellung volumetrischer Daten von zwei verschiedenen Perspektiven. Kapitel 4 und 5 führten eine hierarchische Datenstruktur ein, die später auch die semi-transparente Darstellung erlaubte. Es zeigte sich, dass dies nur effizient gelöst werden kann, wenn auftretende inhomogene Regionen während des hierarchischen Durchlaufens des Kd-Baum durch einen Gittertraversierer abgearbeitet werden. Kapitel 6 beschrieb andererseits einen inkrementiellen Traversierer, der um eine räumliche Hierarchie ergänzt werden müsste, um eine effiziente Isoflächen-Darstellung zu ermöglichen. Die Erweiterung beider Ansätze mit dem jeweils fehlenden Teil würde deshalb zu einem flexiblen und interaktiven Volumen-Ray-Tracing-System führen, dass im Einführungkapitel beschrieben wurde.

Bibliography

[Amanatides87] *John Amanatides and Andrew Woo.* A Fast Voxel Traversal Algorithm for Ray Tracing. In *EG '87: Proceedings of Eurographics*, pages 3–10, 1987.

[Appel68] *Arthur Appel.* Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pages 37–49, 1968.

[Avila92] *Ricardo S. Avila, Lisa M. Sobierajski, and Arie E. Kaufman.* Towards a Comprehensive Volume Visualization System. In *VIS '92: Proceedings of the 3th IEEE Visualization*, pages 13–20, 1992.

[Banks03] *David C. Banks and Stephen Linton.* Counting Cases in Marching Cubes: Toward a Generic Algorithm for Producing Substitopes. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 51–58, 2003.

[Bennett01] *Janine Bennett, Richard Cook, Nelson Max, Deborah May, and Peter Williams.* Parallelizing a High Accuracy Hardware-assisted Volume Renderer for Meshes with Arbitrary Polyhedra. In *PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and large-data Visualization and Graphics*, pages 101–106, 2001.

[Benthin04] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* Interactive Ray Tracing of Free-Form Surfaces. In *Afrigraph '04: Proceedings of the 2th International Conference on Computer Graphics, Virtual Reality and Interaction in Africa*, pages 99–106, 2004.

[Benthin06] *Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich.* Ray Tracing on the CELL Processor. In

RT '06: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pages 15–23, 2006.

[Bhaniramka73] *Praveen Bhaniramka, Rephael Wenger, and Roger Crawfis.* Isosurfacing in higher Dimensions. In *VIS '00: Proceedings of the 11th IEEE Visualization*, page 2000, 267–273.

[Brauchle06] *Roman Brauchle.* Realtime Visualization of Time-Varying Volume Data. Master's thesis, Computer Graphics Group, Saarland University, 2006.

[Bunyk97] *Paul Bunyk, Arie E. Kaufman, and Cláudio T. Silva.* Simple, Fast, and Robust Ray Casting of Irregular Grids. In *Dagstuhl '97: Scientific Visualization*, pages 30–36, 1997.

[Cabral94] *Brian Cabral, Nancy Cam, and Jim Foran.* Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware. In *VVS '94: Proceedings of the 1994 IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[Chen04] *Wei Chen, Liu Ren, Matthias Zwicker, and Hanspeter Pfister.* Hardware-Accelerated Adaptive EWA Volume Splatting. In *VIS '04: Proceedings of the 15th IEEE Visualization*, pages 67–74, 2004.

[Choi02] *Sunghee Choi.* The Delaunay Tetrahedralization from Delaunay Trangulated Surfaces. In *SCG '02: Proceedings of the 8th Symposium on Computational Geometry*, pages 145–150, 20002.

[Cignoni96] *Paolo Cignoni, Claudio Montani, Enrico Puppo, and Roberto Scopigno.* Optimal Isosurface Extraction from Irregular Volume Data. In *VVS '96: Proceedings of the 1996 Symposium on Volume Visualization*, pages 31–38, 1996.

[Cohen94] *Daniel Cohen and Zvi Sheffner.* Proximity clouds - an acelration technique for 3D grid traversal. *The Visual Computer*, pages 27–38, 1994.

[Cohen03] *Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen.* Wang Tiles for Image and Texture Generation. *ACM Transactions on Graphics*, Vol. 22, No. 3, pages 287–294, 2003.

[Danskin92] *John Danskin and Pat Hanrahan.* Fast Algorithms for Volume Ray Tracing. In *VVS '92: Proceedings of the 1992 Symposium on Volume Visualization*, pages 91–98, 1992.

[de Berg00] *Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational Geometry. Algorithms and Applications.* Springer, 2000.

[DeMarle03] *David E. DeMarle, Stive Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen.* Distributed Interactive Ray Tracing for Large Volume Visualization. In *PVG '03: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[Deussen05] *Oliver Deussen and Bernd Lintermann. Digital Design of Nature – Computer Generated Plants and Organics.* Springer, 2005.

[Dietrich03] *Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek.* The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *OpenSG '03: Proceedings of the OpenSG Symposium*, pages 23–31, 2003.

[Dietrich05a] *Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek.* Realistic and Interactive Visualization of High-Density Plant Ecosystems. In *Proceedings of the Eurographics Workshop on Natural Phenomena*, pages 73–81, 2005.

[Dietrich05b] *Andreas Dietrich, Ingo Wald, and Philipp Slusallek.* Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In *VMV '05: Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization*, pages 303–310, 2005.

[Dietrich06] *Andreas Dietrich, Gerd Marmitt, and Philipp Slusallek.* Terrain Guided Multi-Level Instancing of Highly Complex Plant Populations. In *RT '06: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 169–176, 2006.

[Drebin88] *Robert A. Drebin, Loren Carpenter, and Pat Hanrahan.* Volume Rendering. In *SIGGRAPH '88: Proceedings of the 15th annual Conference on Computer Graphics and Interactive Techniques*, pages 65–74, 1988.

[Ellsworth00] *David Ellsworth, Ling-Jen Chiang, and Han-Wei Shen.* Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics. In *VVS '00: Proceedings of the 2000 Symposium on Volume Visualization*, pages 119–128, 2000.

[Engel01] *Klaus Engel, Martin Kraus, and Thomas Ertl.* High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*, pages 9–16, 2001.

[Engel06] *Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christoph Rezk-Salama, and Daniel Weiskopf. Real-Time Volume Graphics.* A K Peters, Ltd., 2006.

[Erickson97] *Jeff Erickson.* Pluecker Coordinates. *Ray Tracing News*, Vol. 10, No. pages 3, 1997.

[Freund97] *Jason Freund and Kenneth Sloan.* Accelated Volume Rendering Using Homogeneous Regions Encoding. In *VIS '97: Proceedings of the 8th IEEE Visualization*, pages 191–196, 1997.

[Friedrich07] *Heiko Friedrich, Ingo Wald, Johannes Günther, Gerd Marmitt, and Philipp Slusallek.* Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In *EGPGV '07: Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 109–116, 2007.

[Frühauf94] *Thomas Frühauf.* Raycasting of Nonregularly Structured Volume Data. In *EG '94: Proceedings of Eurographics*, pages 295–303, 1994.

[Gao03] *Jinzhu Gao, Jian Huang, Han-Wei Shen, and James Arthur Kohl.* Visibility Culling Using Plenoptic Opacity Functions for Large Volume Visualization. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 341–348, 2003.

[Gao04] *Jinzhu Gao, Han-Wei Shen, Jian Huang, and James Arthur Kohl.* Visibility Culling for Time-Varying Volume Rendering Using Temporal Occlusion Coherence. In *VIS '04: Proceedings of the 15th IEEE Visualization*, pages 147–154, 2004.

[Garrity90] *Michael P. Garrity.* Raytracing Irregular Volume Data. In *VVS '90: Proceedings of the 1990 Symposium on Volume Visualization*, pages 35–40, 1990.

[Glassner95] *Andrew Glassner. Principles of Digital Image Synthesis.* Morgan Kaufmann, 1995.

[Gouraud71] *Henry Gouraud.* Continuous Shading of Curved Surfaces. *Communications of the ACM*, Vol. 18, No. 6, pages 623–629, 1971.

[Grimm04] *Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Meister Eduard Gröller.* Memory Efficient Acceleration Structures and Techniques for CPU-based Volume Raycasting of Large Data. In *VOLVIS '04: Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, 2004.

[Groß07] *Matthias Groß, Carsten Lojewski, Martin Bertram, and Hans Hagen.* Fast Implicit Kd-trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for large Scalar Fields. In *CGIM '07: Proceedings of Computer Graphics and Imaging*, pages 67–74, 2007.

[Günther04] *Johannes Günther, Ingo Wald, and Philipp Slusallek.* Realtime Caustics using Distributed Photon Mapping. In *EGRW '04: Proceedings of the 15th Eurographics Workshop on Rendering*, pages 111–121, 2004.

[Guthe02] *Stefan Guthe, Stefan Roettger, Andreas Schieber, Wolfgang Strasser, and Thomas Ertl.* High-Quality Unstructured Volume Rendering on the PC Platform. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 1–8, 2002.

[Hadwiger05] *Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross.* Real-TimeRay-Casting

and Advanced Shading of Discrete Isosurfaces. In *EG '05: Proceedings of Eurographics*, pages 303–312, 2005.

[Haines94] *Eric Haines.* Point in Polygon Strategies. In *Graphics Gems IV*, pages 24–46. Academic Press, 1994.

[Havran01] *Vlastimil Havran.* Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Heckbert89] *Paul Heckbert.* Fundamentals of Texture Mapping and Image Warping. Master's thesis, Department of Electrical Engineering and Computer Science, University of California, 1989.

[Hege93] *Hans-Christian Hege, Tobias Höllerer, and Detlev Stalling.* Volume Rendering - Mathematical Models and Algorithmic Aspects. Technical report, ZIB (Konrad-Zuse-Zentrum), 1993.

[Hong98] *Lichan Hong and Arie Kaufman.* Accelerated Ray-casting for Curvilinear Volumes. In *VIS '98: Proceedings of the 9th IEEE Visualization*, pages 247–253, 1998.

[Hong99] *Lichan Hong and Arie E. Kaufman.* Fast Projection-Based Ray-Casting Algorithm for Rendering Curvilinear Volumes. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 4, pages 322–332, 1999.

[Hong05] *Wei Hong, Feng Qiu, and Arie Kaufman.* GPU-based Object-Order Ray-Casting for Large Datasets. In *VG '05: Proceedings of the International Workshop on Volume Graphics*, pages 177–186, 2005.

[Hounsfield80] *Godfrey N. Hounsfield.* Nobel Award address. Computed medical imaging. *Medical Physics*, Vol. 7, No. 4, pages 283–290, 1980.

[Intel] Intel Pentium III Streaming SIMD Extensions. http://-developer.intel.com/vtune/cbts/simd.htm.

[Jang04] *Yun Jang, Manfred Weiler, Matthias Hopf, Jingshu Huang, David S. Ebert, Kelly P. Gaither, and Thomas Ertl.* Interactively Visualizing Procedurally Encoded Scalar Fields. In

VISSYM '04: Proceedings of EG/IEEE TCVG Symposium on Visualization, pages 35–44, 2004.

[Jensen96] *Henrik Wann Jensen.* Global Illumination using Photon Maps. In *EGRW '96: Proceedings of the 13th Eurographics Workshop on Rendering*, pages 21–30, 1996.

[Kajiya86] *James T. Kajiya.* The Rendering Equation. In *SIGGRAPH '86: Proceedings of the 13th annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, 1986.

[Kay79] *Douglas Scott Kay and Donald Greenberg.* Transparency for Computer Synthesized Images. *SIGGRAPH Comput. Graph.*, Vol. 13, No. 2, pages 158–164, 1979.

[Klein04] *Thomas Klein, Siman Stegmaier, and Thomas Ertl.* Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[Knittel97] *Günter Knittel and Wolfgang Strasser.* VIZARD - Visualization Accelerator for Realtime Display. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 139–146, 1997.

[Knittel00] *Günter Knittel.* The ULTRAVIS System. In *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 71–79, 2000.

[Knoll06] *Aaron Knoll, Ingo Wald, Steven Parker, and Charles Hansen.* Interactive Isosurface Ray Tracing of Large Octree Volumes. In *RT '06: IEEE Symposium on Interactive Ray Tracing 2006*, pages 115–124, 2006.

[Kreeger99] *Kevin Kreeger and Arie Kaufman.* Hybrid Volume and Polygon Rendering with Cube Hardware. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 15–24, 1999.

[Krüger03] *J. Krüger and R. Westermann.* Acceleration Techniques for GPU-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 287–292, 2003.

[Lacroute94] *Philippe Lacroute and Marc Levoy.* Fast Volume Rendering using a Shear-Warp Factorization of the viewing Transformation. In *SIGGRAPH '94: Proceedings of the 21st annual Conference on Computer Graphics and Interactive Techniques*, pages 451–458, 1994.

[Lakare04] *Sarang Lakare and Arie Kaufman.* Light Weight Space Leaping using Ray Coherence. In *VIS '04: Proceedings of the 15th IEEE Visualization*, pages 19–26, 2004.

[Levoy90a] *Marc Levoy.* Efficient Ray Tracing for Volume Data. *ACM Transactions on Graphics*, Vol. 9, No. 3, pages 245–261, 1990.

[Levoy90b] *Marc Levoy.* Volume Rendering. *IEEE Coomputer Graphics Applications*, Vol. 10, No. 2, pages 33–40, 1990.

[Li03] *Wei Li, Klaus Müller, and Arie Kaufman.* Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 317–324, 2003.

[Lorensen87] *William E. Lorensen and Harvey E. Cline.* Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual Conference on Computer Graphics and Interactive Techniques*, pages 163–169, 1987.

[Lucas92] *Bruce Lucas.* A Scientific Visualization Renderer. In *VIS '92: Proceedings of the 3rd IEEE Visualization*, pages 227–234, 1992.

[Ma95] *Kwan-Liu Ma.* Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *PRS '95: Proceedings of the IEEE Symposium on Parallel rendering*, pages 23–30, 1995.

[Ma97] *Kwan-Liu Ma and Thomas W. Crockett.* A Scalable Parallel Cell-Projection Volume Rendering Algorithm for three-dimensional Unstructured Data. In *PRS '97: Proceedings of the IEEE Symposium on Parallel Rendering*, pages 95–104, 1997.

[MacDonald89] *David J. MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. In *GI '89: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Interface*, pages 152–63, 1989.

[Marmitt02] *Gerd Marmitt and Andrew T. Duchowski.* Modeling Visual Attention in VR: Measuring the Accuracy of Predicted Scanpaths. In *EG '02: Proceedings of Eurographics Short Presentations*, pages 217–226, 2002.

[Marmitt04] *Gerd Marmitt, Andreas Kleer, Ingo Wald, Heiko Friedrich, and Philipp Slusallek.* Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *VMV '04: Proceedings of 9th International Fall Workshop - Vision, Modeling, and Visualization*, pages 429–435, 2004.

[Marmitt05] *Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek.* Recent Advancements in Ray-Tracing based Volume Rendering Techniques. In *VMV '05: Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization*, pages 131–138, 2005.

[Marmitt06a] *Gerd Marmitt, Roman Brauchle, Heiko Friedrich, and Philipp Slusallek.* Accelerated and Extended Building of Implicit Kd-Trees for Volume Ray Tracing. In *VMV '06: Proceedings of 11th International Fall Workshop - Vision, Modeling, and Visualization*, pages 317–324, 2006.

[Marmitt06b] *Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek.* Interactive Volume Rendering with Ray Tracing. In *EG '06: Proceedings of Eurographics State of the Art Reports*, pages 115–136, 2006.

[Marmitt06c] *Gerd Marmitt and Philipp Slusallek.* Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *EUROVIS '06: Proceedings of the EG/IEEE Symposium on Data Visualisation*, pages 131–138, 2006.

[Marmitt08] *Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek.* Efficient CPU-based Volume Ray Tracing. *Computer Graphics Forum*, Vol. 27, No. 6, pages 1687–1709, 2008.

[Marschner94] *Stephen R. Marschner and Richard J. Lobb.* An Evaluation of Reconstruction Filters for Volume Rendering. In *VIS*

'94: Proceedings of the 5th IEEE Visualization, pages 100–107, 1994.

[McDonnell04] *Kevin T. McDonnell, Yu-Sung Chang, and Hong Qin.* Interpolatory, solid subdivision of unstructured hexahedral meshes. *The Visual Computer*, Vol. 20, No. 6, pages 418–436, 2004.

[Meissner00] *Michael Meissner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis.* A Practical Evaluation of Popular Volume Rendering Algorithms. In *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 81–90, 2000.

[Mirin99] *Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, William P. Dannevik, Andris, M. Dimits, Mark A. Duchaineau, D. E. Eliason, Daniel R. Schikore, S. E. Anderson, D. H. Porter, and Paul R. Woodward.* Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. In *Supercomputing '99: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 70–78, 1999.

[Mora02] *Benjamin Mora, Jean Pierre Jessel, and Rene Caubet.* A new Object-order Ray-Casting Algorithm. In *VIS '02: Proceedings of the 13th IEEE Visualization*, pages 203–210, 2002.

[Müller96] *Klaus Müller and Roni Yagel.* Fast Perspective Volume Rendering with Splatting by Utilizing a Ray-driven Approach. In *VIS '96: Proceedings of the 7th conference on Visualization*, pages 65–72, 1996.

[Müller98] *Klaus Müller and Roger Crawfis.* Eliminating Popping Artifacts in Sheet Buffer-Based Splatting. In *VIS '98: Proceedings of the 9th IEEE Visualization*, pages 239–245, 1998.

[Müller06] *Christoph Müller, Magnus Strengert, and Thomas Ertl.* Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *EGPGV '06: Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 59–66, 2006.

[Neophytou06] *Neophtos Neophytou, Klaus Mueller, Kevin T. McDonnel, Wei Hong, Xin Guan, Hong Qin, and Arie Kaufmann.* GPU-Accelerated Volume Splatting with Elliptical RBFs. In *EUROVIS '06: Proceedings of the EG/IEEE Symposium on Data Visualisation*, pages 13–20, 2006.

[Neubauer02] *André Neubauer, Lukas Mroz, Helwig Hauser, and Rainer Wegenkittl.* Cell-Based First-Hit Ray Casting. In *EUROVIS '02: Proceedings of the EG/IEEE Symposium on Data Visualisation*, pages 77–86, 2002.

[Nielson90] *G.M. Nielson and B. Haman.* Techniques for the Interactive Visualization of Volumetric Data. In *VIS '90: Proceedings of the 1st IEEE Visualization*, pages 45–50, 1990.

[Nielson03] *Gregory M. Nielson.* MC*: Star Functions for Marching Cubes. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003*, pages 59–66, 2003.

[Nielson04] *Gregory M. Nielson.* Dual Marching Cubes. In *VIS '04: Proceedings of the 15th IEEE Visualization*, pages 489–496, 2004.

[nvidia] CUDA Programming Guide 1.1. http://-developer.download.nvidia.com/cuda/1_1/-NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[Oppenheim75] *Alan V. Oppenheim and Ronald W. Schafer. Digital Signal Processing.* Prentice Hall, 1975.

[Parker99a] *Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen.* Interactive Ray Tracing. In *SI3D '99: Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

[Parker99b] *Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley.* Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 3, pages 223–250, 1999.

[Pascucci00] *Valerio Pascucci and Chandrajit L. Bajaj.* Time Critical Isosurface Refinement And Smoothing. In *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 33–42, 2000.

[Pascucci04] *Valerio Pascucci.* Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *VISSYM '99: Proceedings of EG/IEEE TCVG Symposium on Visualization*, pages 293–300, 2004.

[Pfister99] *Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler.* The VolumePro Real-Time Ray-casting System. In *SIGGRAPH '99: Proceedings of the 26th annual Conference on Computer Graphics and Interactive Techniques*, pages 251–260, 1999.

[Phong75] *Bui Tuong Phong.* Illumination for Computer Generated Images. *Communications of the ACM*, Vol. 18, No. 6, pages 311–317, 1975.

[Platis03] *Nikos Platis and Theoharis Theoharis.* Fast Ray-Tetrahedron Intersection Using Plücker Coordinates. *Journal of Graphics Tools*, Vol. 8, No. 4, pages 37–48, 2003.

[Pomi03] *Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek.* Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments. In *VMV '03: Proceedings of 8th International Fall Workshop - Vision, Modeling, and Visualization*, pages 261–269, 2003.

[Purcell02] *Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan.* Ray Tracing on Programmable Graphics Hardware. In *SIGGRAPH '02: Proceedings of the 29th annual Conference on Computer Graphics and Interactive Techniques*, pages 703–712, 2002.

[Ramsey04] *Shaun Ramsey, Kristin Potter, and Charles Hansen.* Ray Bilinear Patch Intersections. *Journal of Graphics Tools*, Vol. 9, No. 3, pages 41–47, 2004.

[Reinhard02] *Erik Reinhard, Charles Hansen, and Steve Parker.* Interactive Ray Tracing of Time Varying Data. In *EGPGV '02: Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization*, pages 77–82, 2002.

[Reshetov05] *Alexander Reshetov, Alexei Soupikov, and Jim Hurley.* Multi-level Ray Tracing Algorithm. In *SIGGRAPH '05:*

Proceedings of the 32th annual Conference on Computer Graphics and Interactive Techniques, pages 1176–1185, 2005.

[Rössl04] *Christian Rössl, Frank Zeilfelder, Günther Nürnberger, and Hans-Peter Seidel.* Reconstruction of Volume Data with Quadratic Super Splines. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No. 10, pages 397–409, 2004.

[Röttger00] *Stefan Röttger, Martin Kraus, and Thomas Ertl.* Hardware-Accelerated Volume And Isosurface Rendering Based On Cell-Projection. In *VIS '00: Proceedings of the 11th IEEE Visualization*, pages 109–116, 2000.

[Röttger03] *Stefan Röttger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser.* Smart Hardware-Accelerated Volume Rendering. In *VISSYM '03: Proceedings of EG/IEEE TCVG Symposium on Visualization*, pages 231–238, 2003.

[Schmittler02] *Jörg Schmittler, Ingo Wald, and Philipp Slusallek.* SaarCOR – A Hardware Architecture for Ray Tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 27–36, 2002.

[Schwarze98] *Jochen Schwarze.* Cubic and Quartic Roots. In *Graphics Gems*, pages 404–407. Academic Press, 1998.

[Shen99] *Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma.* A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. In *VIS '99: Proceedings of the 10th IEEE Visualization*, pages 371–377, 1999.

[Shirley90] *Peter Shirley and Allan Tuchman.* A Polygonal Approximation to Direct Scalar Volume Rendering. In *VVS '90: Proceedings of the 1990 Symposium on Volume Visualization*, pages 63–70, 1990.

[Shirley05] *Peter Shirley. Fundamentals of Computer Graphics.* A K Peters, 2005.

[Siddon85] *Robert L. Siddon.* Fast Calculation of the Exact Radiological Path for a Three-Dimensional CT Array. *Medical Physics*, Vol. 12, No. 2, pages 252–255, 1985.

[Stegmaier05] *Simon Stegmaier, Magnus Strengert, Thomas Klein, and Thomas Ertl.* A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *VG '05: Proceedings of the International Workshop on Volume Graphics*, pages 187–195, 2005.

[Strengert06] *Magnus Strengert, Thomas Klein, Ralf Botchen, Siman Stegmaier, Min Chen, and Thomas Ertl.* Spectral Volume Rendering using GPU-based Raycasting. *The Visual Computer*, Vol. 22, No. 8, pages 550–561, 2006.

[Subramanian90] *K. R. Subramanian and Donald S. Fussel.* Applying Space Subdivision Techniques to Volume Rendering. In *VIS '90: Proceedings of the 1st IEEE Visualization*, pages 150–159, 1990.

[Sweeney02] *Jon Sweeney and Klaus Müller.* Shear-Warp Deluxe: The Shear-Warp Algorithm Revisited. In *VISSYM '02: Proceedings of EG/IEEE TCVG Symposium on Visualization*, pages 95–104, 2002.

[Theisel01] *Holger Theisel.* CAGD and Scientific Visualization. PhD thesis, Faculty of Electrical Engineering, Rostock University, 2001. Habilitionsschrift.

[Udupa93] *Jayaram K. Udupa and Dewey Odhner.* Shell Rendering. *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, pages 58–67, 1993.

[VHP] The Visible Human Project. http://www.nlm.nih.gov/-research/visible/visible_human.html.

[Wald01a] *Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek.* Interactive Rendering with Coherent Ray Tracing. In *EG '02: Proceedings of Eurographics*, pages 153–164, 2001.

[Wald01b] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models.

In *EGRW '01: Proceedings of the 12th Eurographics Workshop on Rendering*, pages 274–285, 2001.

[Wald02a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Computer Graphics Lab, Saarland University, 2002.

[Wald02b] *Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek.* Interactive Global Illumination using Fast Ray Tracing. In *EGRW '02: Proceedings of the 13th Eurographics Workshop on Rendering*, pages 15–24, 2002.

[Wald04a] *Ingo Wald.* Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wald04b] *Ingo Wald, Andreas Dietrich, and Philipp Slusallek.* An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *EGRW '04: Proceedings of the 15th Eurographics Workshop on Rendering*, pages 81–92, 2004.

[Wald05] *Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel.* Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 11, No. 5, pages 562–573, 2005.

[Wald07] *Ingo Wald, Heiko Friedrich, Aaron Knoll, and Charles Hansen.* Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. pages *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[Wang61] *Hao Wang.* Proving Theorems by Pattern Recognition. *Bell Systems Technical Journal*, Vol. 40, No. ???, pages 1–42, 1961.

[Weghorst84] *Hank Weghorst, Gary Hooper, and Donald P. Greenberg.* Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, Vol. 3, No. 1, pages 52–69, 1984.

[Weiler03] *Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl.* Hardware-Based Ray Casting for Tetrahedral Meshes. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 333–340, 2003.

[Weiler04] *Manfred Weiler, Paula N. Mallon, Martin Kraus, and Thomas Ertl.* Texture-Encoded Tetrahedral Strips. In *VV '04: Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, pages 71–78, 2004.

[Westover90] *Lee Westover.* Footprint Evaluation for Volume Rendering. In *SIGGRAPH '90: Proceedings of the 17th annual Conference on Computer Graphics and Interactive Techniques*, pages 367–376, 1990.

[Whitted80] *Turner Whitted.* An Improved Illumination Model for Shaded Display. *CACM*, Vol. 23, No. 6, pages 343–349, 1980.

[Wilhelms90] *Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri.* Direct Volume Rendering of CurvilinearVolumes. In *VVS '90: Proceedings of the 1990 Symposium on Volume Visualization*, pages 41–47, 1990.

[Wilhelms92] *Jane Wilhelms and Allen Van Gelder.* Octrees for faster Isosurface Generation. *ACM Transactions on Graphics*, Vol. 11, No. 3, pages 201–227, 1992.

[Wilhelms96] *Jane Wilhelms, Allen Van Gelder, Paul Tarantino, and Jonathan Gibbs.* Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *VIS '96: Proceedings of the 7th IEEE Visualization*, pages 57–64, 1996.

[Williams92] *Peter L. Williams.* Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, Vol. 11, No. 2, pages 103–126, 1992.

[Williams98] *Peter L. Williams, Nelson L. Max, and Clifford M. Stein.* A High Accuracy Volume Renderer for Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No. 1, pages 37–54, 1998.

[Woodring03] *Jonathan Woodring, Chaoli Wang, and Han-Wei Shen.* High Dimensional Direct Rendering of Time-Varying Volumetric Data. In *VIS '03: Proceedings of the 14th IEEE Visualization*, pages 417–424, 2003.

[Woop05] *Sven Woop, Joerg Schmittler, and Philipp Slusallek.* RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *SIGGRAPH '05: Proceedings of the 32th annual Conference on Computer Graphics and Interactive Techniques*, pages 434–444, 2005.

[Woop06] *Sven Woop, Gerd Marmitt, and Philipp Slusallek.* B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *HWWS '06: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 67–77, 2006.

[Wu03] *Yin Wu, Vishal Bhatia, Hugh Lauer, and Larry Seiler.* Shear-image Order Ray Casting Volume Rendering. In *SI3D '03: Proceedings of the Symposium on Interactive 3D graphics*, pages 152–162, 2003.

[Yagel94] *Roni Yagel.* Shell Accelerated Volume Rendering of Transparent Regions. *The Visual Computer*, Vol. 10, No. 1, pages 53–61, 1994.

[Younesy05] *Hamid Younesy, Torsten Möller, and Hamish Carr.* Visualization of Time-Varying Volumetric Data using Differential Time-Histogram Table. In *VG: '05: Proceedings of Volume Graphics*, pages 21–29, 2005.

[Zhang97] *Hansong Zhang, Dinesh Manocha, Tom Hudson, and III Kenneth E. Hoff.* Visibility Culling using Hierarchical Occlusion Maps. In *SIGGRAPH '97: Proceedings of the 24th annual Conference on Computer Graphics and Interactive Techniques*, pages 77–88, 1997.

[Zwicker01] *Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross.* EWA Volume Splatting. In *VIS '01: Proceedings of the 12th IEEE Visualization*, pages 29–36, 2001.