Volumetric Methods for the Real-Time Display of Natural Gaseous Phenomena

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Stefan Roettger

aus Erlangen

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. M. Stamminger
	Prof. Dr. W. Heidrich

Tag der mündlichen Prüfung: 2. Juni 2004

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2004

Contents

1 Motivation and Outline **8**

2 The OpenGL Rendering Pipeline **11**
2.1 Basic Layout of the Rendering Pipeline 11
2.2 Rendering Example 13
2.3 Lighting and Texturing 15
2.3.1 Direct Lighting 16
2.3.2 Texture Mapping 19
2.4 Programmable Graphics Hardware 21
2.4.1 Vertex Shaders 22
2.4.2 Pixel Shaders and Fragment Programs 23

3 A Brief History of Terrain Rendering **26**
3.1 Data Representation 26
3.2 TINs 26
3.3 S-LOD 28
3.4 Progressive Meshes 28
3.5 C-LOD Algorithms 29
3.5.1 Lindstrom's Algorithm 30
3.5.2 Duchaineau's Algorithm 33
3.5.3 Roettger's Algorithm 35
3.6 Future Development 39

4 The Terrain Rendering Pipeline **40**
4.1 Landscape Data Generation 41
4.2 Real Time Display of the Terrain 41
4.3 Terrain Material 42
4.4 Terrain Illumination 42
4.5 Organic Features 43
4.6 Global Volumetric Effects 43
4.7 Volumetric Effects in Practice 44

5 Natural Gaseous Phenomena **46**
5.1 Sky Dome 47
5.2 OpenGL Fog 47
5.3 Layered Fog 50
5.4 Bounded Layered Fog 51
5.5 Billboards 51
5.6 Metaball Methods 52
5.7 Impostor Based Methods 53

6 Volume Rendering: The Basics **57**
6.1 Basic Principles 57
6.2 The Rendering Equation 57
6.3 The Ray Integral 59
6.4 Light Scattering in Participating Media 60
6.5 Rendering Solutions for Participating Media 61

7 Direct Volume Rendering **63**
7.1 Transfer Functions 63
7.2 Grid Types 64
7.3 Ray Casting 64
7.4 Slicing via 3D Textures 66

8 Pre-Integrated Cell-Projection **68**
8.1 Visibility Sorting 68
8.2 The Original PT Algorithm 69
8.3 Drawbacks of the Original PT Algorithm 70
8.4 PT with Accurate Opacity and Color 71
8.5 A New Approximation for PT 74
8.6 Prior Work about Isosurfaces 75
8.7 Hardware-Accelerated Marching Cells 76
8.8 Flat-Shaded Isosurfaces 77
8.9 Smoothly Shaded Isosurfaces 79
8.10 Colored and Multiple Isosurfaces 80
8.11 Mixing Isosurfaces with Projected Volumes 81
8.12 Performance Comparison 83
8.13 Heaviside Excursion 84
8.14 Pre- vs. Post-Classification 85

9 Unstructured Volume Rendering on the PC **88**
9.1 High Resolution Ray Integral 88
9.1.1 Opacity Reconstruction 88

9.1.2 Chromaticity Reconstruction 90
9.2 Hardware Accelerated Pre-Integration 91
9.3 Performance Measurements 95

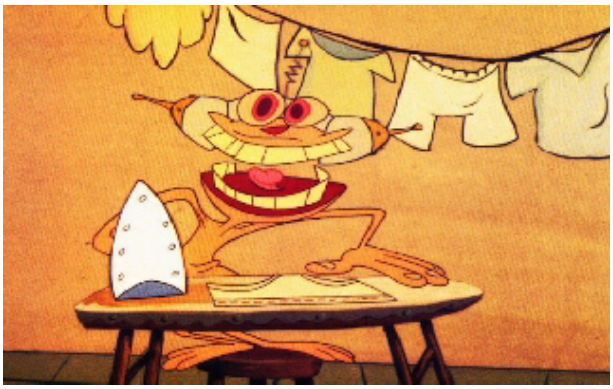
10 Ground Fog Rendering **98**
10.1 Theoretical Performance 99
10.2 Practical Performance Analysis 99
10.3 Projected Convex Polyhedra Algorithm (PCP) 100
10.4 Applications of the PCP Method 102

11 Cloud Rendering **106**
11.1 View-Dependent Rendering 106
11.2 C-LOD Rendering 107
11.3 Generating Continuous Levels of Detail 107
11.3.1 Hierarchical Volume Representation 107
11.3.2 View-Dependent Mesh Simplification 107
11.3.3 Building a Conforming Mesh 109
11.3.4 Hierarchical Error Propagation 109
11.4 Volumetric Morphing 110
11.5 Cell Projection 112
11.6 Non-Photorealistic Cloud Rendering 113
11.6.1 Modified PT Algorithm 114
11.6.2 Non-Photorealistic Lighting 114
11.7 Performance Measurements 115
11.8 Discussion 116

12 Summary **118**
12.1 Decision Chart 120
12.2 Availability and Licensing 121

13 Zusammenfassung **140**

Acknowledgements



...I’ll just lie back and think pleasant thoughts. Chicken pot pie... Chocolate covered RAIsins... EEGlazed Hammmmm... They think I’m crazy... But I know better... It is not I who am crazy... It is I who am MAD!... Didn’t you hear ’em? Didn’t you see the crowds?!! Oh my beloved ice cream bar... How I love to lick your creamy center... eeyaaarghruch... eeyaarghrunch... eeyaarghrunch... And your oh-so-nutty chocolate covering... You’re not like the others... You like the same things I do... Wax paper... Boiled football leather... Dog breath... WE’RE NOT HITCHHIKING ANYMORE... WE’RE RIDING...

Ren Höek’s insane ravings in “Space Madness”

First of all I would like to thank Martin Kraus of VIS, Stefan Guthe of WSI/GRIS, and Ingo Frick of Massive Development for the great teamwork and all others from the computer graphics labs in Erlangen, Tübingen, and Stuttgart for their support. They all have been a great help in writing this thesis.

On 7th February of 2002 I was lucky to become the father of my daughter Leonie. I have to say the warmest thanks to my girl friend who so much cared about her while I was busy writing, so this thesis is dedicated to my dear Leonie and Petra!

Stuttgart / Erlangen, 9th September 2004

Stefan Röttger

Abstract

Blaise Pascal: *The last thing one knows*
in constructing a work
is what to put first.

Compared to the nineties where fast 3D graphics was the domain of expensive workstations, in the last few years the development of ever faster 3D graphics hardware was mainly driven by the gaming industry. The upcoming of programmable PC graphics hardware has opened the field for new graphics algorithms which allow unprecedented realism in real time applications. Nevertheless, one area of application is persistently resisting most efforts to achieve sufficient rendering performance: This is the area of volume rendering. Because of the huge amounts of data that have to be processed to obtain a three-dimensional visualization, it is very challenging to achieve real time performance for large volumetric data sets.

In this thesis we try to tackle this problem in one specific application field. We devise algorithms that are suitable for the real time display of natural gaseous phenomena. In particular our goal is to render clouds and fog in real time. In principle, the problem reduces to solving the so called ray integral. A common technique for solving this ray integral is ray casting which collects the incoming light on each viewing ray by sampling the volume. On the one hand ray casting achieves very good rendering quality, but on the other hand it becomes very slow at high screen resolutions. Many improvements have been presented to accelerate the original approach, but despite all efforts ray casting is still only beginning to be an option for high-quality real-time rendering. Very recent advances in graphics hardware have lead to the implementation of hardware-accelerated ray casters, but this approach still suffers from a variety of limitations of the graphics hardware.

The main technique developed in this thesis is the so called pre-integrated cell-projection method which offloads as much computation of the ray integral as possible into a preprocessing step. This is the first step toward real-time rendering of natural gaseous phenomena. In a second step we develop a hierarchical approximation scheme which decimates the huge amount of data in a view-dependent way. For this purpose we borrow ideas from the area of terrain rendering and apply the so-called continuous level of detail method to the three-dimensional case, that is fog and cloud volumes. In combination with the pre-integrated cell-projection method this permits real-time flights through natural looking clouds and ground fog. In comparison to previous methods image quality is also improved significantly.

Chapter 1
Motivation and Outline

Three-dimensional computer graphics plays an important role in computational science and engineering. The interactive visualization of simulation data and measurements allows to explore experimental results in an very intuitive way. While interactive 3D graphics has been the domain of expensive graphics workstations in the past, the influence of the computer gaming industry is growing with the demand for much more realistic and ever faster 3D graphics in interactive entertainment. As a result, the graphics hardware developed for 3D computer games has become increasingly powerful and cheap and thus has been made available for a broad class of consumers.

The good price performance ratio of consumer graphics cards also opens new fields of application both for scientific visualization and computer games. In this thesis we exploit the feature set of modern PC graphics cards in the specific area of volume rendering. We develop algorithms that are able to display natural gaseous phenomena, which are inherently volumetric, with the focus on real-time rendering. Our decision to focus on real-time algorithms was influenced by a close cooperation with the 3D gaming industry where speed is the one and only imperative. During a cooperation with Massive Development [65] we included our previously developed terrain renderer [89] into their Krass game engine.

In the process of this cooperation we noticed that truly volumetric effects actually are used very rarely in computer games. What is more, most of the applied volumetric effects like explosions and smoke are just fake in a scientific sense. Natural gaseous phenomena like true clouds or fog are missing almost completely due to the inherent complexity of these volumetric effects. In the case that these effects show up in computer games they are mostly implemented using very much simplified physical models which do not match the complex situation in the real world. Another approach encountered frequently in computer games is to use a cloud photo painted on the sky hemisphere. But in this case the possible view points are restricted to a small centric area. The consequence is that one cannot fly through the clouds, for example.

In this thesis we provide solutions for the mentioned problems. We devise algorithms that are suitable for the real-time display of arbitrary natural looking clouds and fog. For this purpose we apply, modify, and extend techniques

known from scientific terrain and volume visualization. In particular we extend the C-LOD algorithm known from terrain rendering to work with scalar volumes and develop the so-called pre-integration technique which allows superior volume rendering quality.

The C-LOD algorithm (acronym for continuous level of detail) takes advantage of the fact that a terrain has a large extent. Thus, for a typical point of view, very far and very near details are viewed at the same time. Since the geometric representation of the terrain consists of triangles, the perspective projection will render some of these triangles very small and some very large. Rather than displaying all triangles in every frame, the C-LOD algorithm repeatedly disposes the smallest triangles until the reduced number of triangles can be handled at real-time. The higher the target frame rate the smaller is the generated number of triangles and the larger the size of the triangles. Since the decimation scheme depends on the view point, roaming over the terrain results in a constantly changing triangulation. This leads to the so-called popping effect, that is a detail suddenly pops up when approaching it. However, if the C-LOD algorithm keeps the triangles popping up below a projected size of one pixel then the popping effect does not become observable.

The C-LOD approximation scheme is not only valid for terrain data, but it can also be applied to other data types. We show that since cloudy skies have a large extent, we can also apply the C-LOD decimation scheme to a volume representing the clouds. In this thesis we describe the necessary adjustments to adapt the C-LOD algorithm to the volumetric case and give examples for rendering clouds and ground fog.

In contrast to terrain data which consists of triangles, the volume needs to be decomposed into tetrahedra. This has a major implication: Other than for triangles, there is no built-in rendering support for tetrahedra as a drawing primitive. In order to render a tetrahedron the so-called projected tetrahedra algorithm has to be applied. It projects the tetrahedra into the viewing plane and transforms each tetrahedron into a set of triangles with associated color and opacity.

In principle, the ray integral has to be solved for every viewing ray intersecting a tetrahedron. The colors and opacities associated with each triangle of the tetrahedral decomposition are only a very rough approximation of this ray integral. For this reason we develop a much more accurate method called pre-integrated cell-projection which exploits the feature set of modern PC graphics hardware. With this solution the appearance of clouds can be modeled much more realistically.

The outline of this thesis is described as follows: As an introduction into interactive computer graphics, we first give an overview of the OpenGL rendering pipeline (Chapter 2). We continue with a chronological survey of the existing terrain rendering algorithms which are the key component of all outdoor engines. Then we proceed with a brief summary of the current state of the art in outdoor

game engine design. This is supposed to illustrate the special demands that are posed on outdoor game engines. To give an actual example of a modern game engine, we describe the graphics engine of the computer game AquaNox [87] (Chapter 4). The terrain renderer of this engine was developed in cooperation with Massive Development at the University of Erlangen in 1998. In the following chapters this terrain rendering engine serves as the framework for rendering realistic clouds and ground fog. The engine also demonstrates the use of simple volumetric effects that are already widespread in computer games (see Figure 4.1). The previous work on visualization of natural gaseous phenomena is presented in Chapter 5. In summary this chapter reveals that efficient volume rendering methods have not yet been applied to display volumetric clouds and fog in real-time. In Chapters 6 and 8 the physical foundation for the desired volumetric effects is developed and basic volume rendering algorithms are introduced (see also [90, 35, 84]). Finally, the fundamental algorithms are put in context, that is they are applied to rendering real fog and clouds in Chapters 10 and 11 using the described C-LOD and pre-integration techniques (see also [86, 85].

Chapter 2
The OpenGL Rendering Pipeline

In recent years, graphics hardware has become a standard equipment of every consumer PC. This development was mainly driven by the gaming industry which has grown to a multi-billion dollar market in only a few years. Due to the tough competition for ever faster graphics hardware the graphics accelerators nowadays include features that not long ago were available only on extremely expensive graphics workstations. Consumer graphics hardware has undergone a huge performance leap but the fundamental operating principle has not changed since the early days when Silicon Graphics dominated the market. This chapter describes the layout of graphics hardware which is designed to accelerate the perspective display of three-dimensional objects and scenes. In comparison to software approaches they achieve speed-ups of several magnitudes and thus are the basic requirement for all types of interactive computer graphics. As a fundamental property the graphics accelerators exploit the enormous potential for parallelization and pipelining of 3D graphics. As a result, the basic processing structure is the so-called rendering pipeline. In the following we describe the main stages of this pipeline and their purpose.

2.1 Basic Layout of the Rendering Pipeline

In general, a three-dimensional scene description has to be converted into a set of graphics primitives before it can be displayed. This process is called tesselation. Typically, graphics accelerators support triangles or convex planar polygons such as quadrilaterals as rendering primitive. Each primitive is defined by a fixed number of 3D vertices and connectivity information together with appearance attributes such as color and texture coordinates. These vertices are passed down the pipeline and processed in the following stages of the pipeline until finally a two-dimensional raster image is computed:

1. **Geometry Processing** transforms the incoming vertices in the 3D spatial domain. Operations like scaling, translation, rotation are performed. Local shading information is computed which is derived from surface normals and a fixed number of light sources. After that the vertices from the stream

are joined together according to their order to form geometric primitives (points, lines, triangles, etc.). Finally the vertices are projected perspectively.

2. **Rasterization** converts the geometric primitives into fragments. Each fragment corresponds to a pixel on the screen. It holds information about depth, color, transparency, and textures of the corresponding pixel.

3. **Fragment Operations** subsequently modify the fragments attributes (e.g. blending). Several tests decide whether a fragment is discarded or displayed on the screen. The Z-test, for example, discards hidden fragments. The final color is written into the frame buffer.

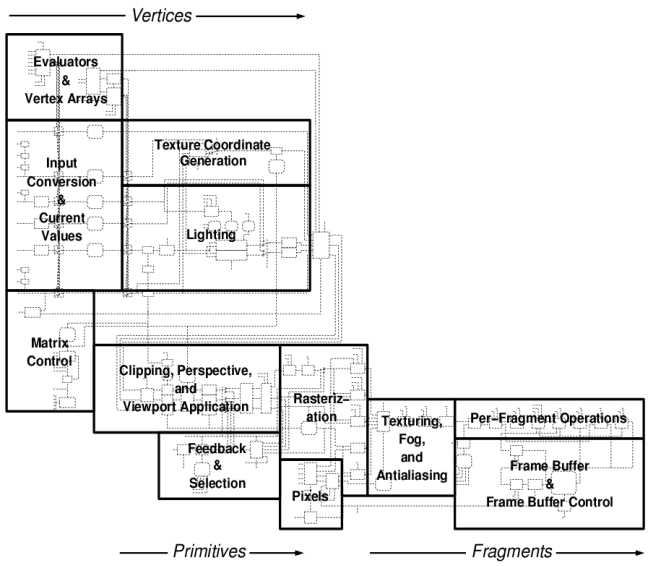


Figure 2.1: The OpenGL 1.1 rendering pipeline.

OpenGL [80, 123] is an open standard which implements all features of the rendering pipeline. It is a software interface with about 200 distinct commands which serve as an abstraction layer between applications and the hardware specific implementation of the pipeline. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 3D graphics API. The specification of the OpenGL standard is guided by an independent consortium, the *OpenGL Architecture Review Board*. The review board ensures the conformity of the specification for all licensed implementations and language bindings. Just like any

software the current version 1.5 has undergone a lot of improvements since its initial version 1.0. The development is pushed by the OpenGL extension mechanism which allows every hardware developer to introduce new technological innovations. Many of these extensions have become a standard in the subsequent release of OpenGL. In contrast to the open structure of OpenGL, Direct3D is a proprietary 3D graphics standard which is only supported on the Windows platform. It is mostly used in the gaming community and for developing multimedia applications. In Figure 2.1 the logical layout of the OpenGL 1.1 rendering pipeline is shown. The blocks dealing with vertices, primitives, and fragments are grouped together.

2.2 Rendering Example

OpenGL is organized as a state engine. Each OpenGL command begins with a ”gl” prefix and changes the content of a state variable or sends vertex data to the graphics processor. A state attribute such as color or texture remains the same until a command is issued that changes it again. So the typical procedure to send triangles down the pipeline is as follows:

- **Set global state**
  `glMatrixMode(GL_PROJECTION);` // modify projection matrix
  `gluPerspective(30,1,1,10);` // initialize perspective projection
  `glEnable(GL_DEPTH_TEST);` // enable Z-test

- **Set attributes**
  `glColor3f(1.0f,1.0f,1.0f);` // change vertex color to white

- **Issue vertices organized as triangles**
  `glBegin(GL_TRIANGLES);` // start primitives
  `glVertex3f(0.0f,1.0f,-5.0f);` // vertex v0
  `glVertex3f(-1.0f,0.0f,-5.0f);` // vertex v1
  `glVertex3f(1.0f,-1.0f,-10.0f);` // vertex v2
  `glColor3f(1.0f,0.0f,0.0f);` // change color to red
  `glVertex3f(1.0f,1.0f,-7.0f);` // vertex v0
  `glColor3f(0.0f,1.0f,0.0f);` // change color to green
  `glVertex3f(-1.0f,-1.0f,-7.0f);` // vertex v1
  `glColor3f(0.0f,0.0f,1.0f);` // change color to blue
  `glVertex3f(1.0f,-1.0f,-7.0f);` // vertex v2
  `glEnd();` // end primitives

This small example renders a small white triangle in front of the viewer, which per default setting is located in the origin and looks downward the negative z-axis. It also renders a smoothly colored triangle which intersects the white triangle (see Figure 2.2). To reconstruct the correct hidden relationship of the two triangles the Z-test must be turned on. Note that the color is specified before the vertices are issued, so that the first three subsequent vertices have the same color, for example.

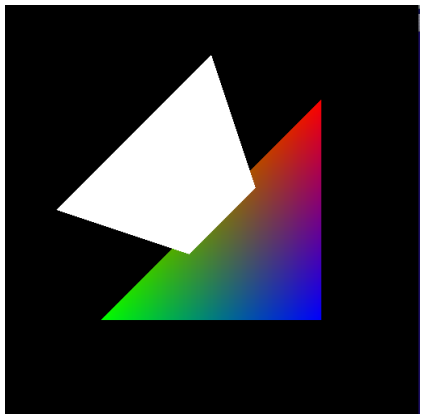


Figure 2.2: An OpenGL rendering example with two triangles that intersect and occlude each other.

The geometric primitives supported by OpenGL are depicted in Figure 2.3, that is mainly points, lines, triangles, quads, triangles strips, quad strips, triangle fans, and planar polygons. Theoretically, current consumer graphics cards are able to process several million to tenth of millions of triangles per second. Since the primitives must be passed to the graphics hardware over a dedicated bus, the practical performance, however, is much less. For this reason, one tries to minimize the number of necessary graphics primitives by a technique called triangle or quad stripping. Typically triangles are part of a mesh. For a regular mesh the vertices of each quadrilateral must be passed four times down the pipeline. By organizing the quadrilaterals in stripes each two new vertices in the vertex stream define a new quadrilateral (compare center case at bottom of Figure 2.3). As a result each vertex must be passed down the pipeline only twice to render the complete mesh.

Vertex arrays are another means of approaching the theoretical performance of graphics accelerators. Vertex arrays are indexed lists of graphics primitives that are stored in dedicated graphics memory, so that the primitives do not need to be transferred through the bottleneck of the bus. If a 3D object entirely fits into graphics memory, vertex arrays are the fastest rendering mode available. The limited amount of dedicated memory usually slows down rendering performance, since data chunks have to be reloaded into graphics memory when needed.

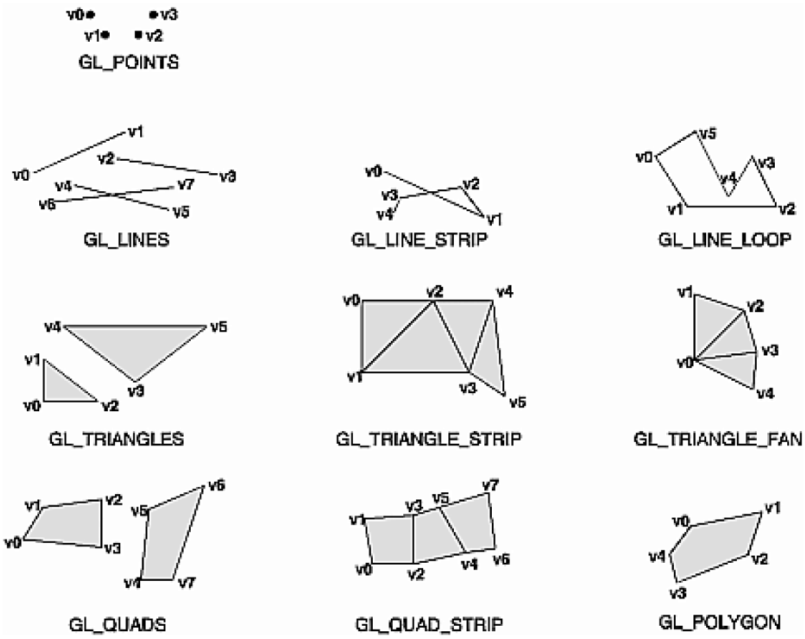


Figure 2.3: The OpenGL rendering primitives (from [123]).

2.3 Lighting and Texturing

Besides the rendering performance the realistic appearance of a three-dimensional object is another prime goal. In the rendering example we have only considered the primary color of an object, but in reality the objects texture and illumination also plays an important role. To demonstrate this Figure 2.4 contrasts the three main contributions to an image: geometry, illumination and texture. The figure shows a famous Mars mountain which is commonly known as the “Face of Mars”. The leftmost image simply depicts the geometry of the mountain by displaying the wire frame of the triangle mesh. At this point the method by which the triangle mesh is created is not relevant. This is explained in full detail in Chapter 3. The next image in the middle shows the mountain as if it were lit by the sun from a near-zenith position. With illumination the basic shape of the mountain is exposed but small details are not yet visible. These are added in the rightmost image where a photograph captured by one of the Mars Viking Orbiters is used as a texture. By adding the texture it becomes clear why the mountain has become famous as the “Face of Mars”. Without the texture it is just a mountain like many others. This fact underlines the importance of texturing in computer graphics.

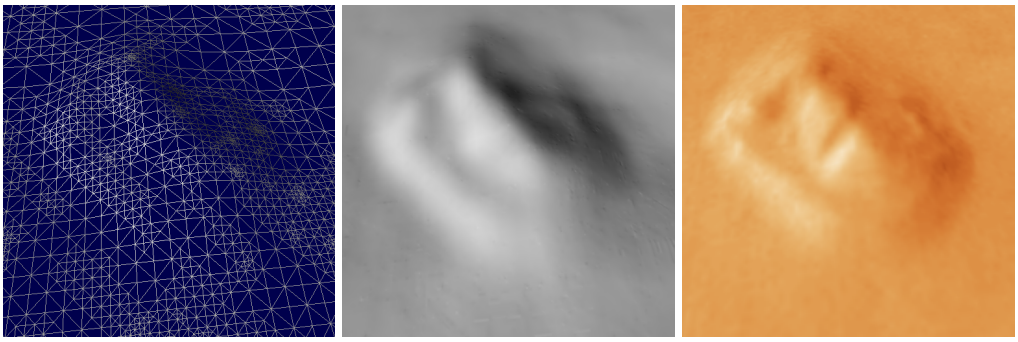


Figure 2.4: Lighting and texturing example. **From left to right**: Wire frame, lit, and textured view of a famous Mars mountain that is commonly known as the "Face of Mars".

2.3.1 Direct Lighting

In the real world the appearance of a scene is the result of an inconceivable amount of photons being reflected from surfaces again and again until they finally reach the observer's eyes. The exact real-time simulation of this process is of course infeasible. While the shear amount of encountered photons prevents the exact reproduction of the illumination, the light reflection properties of surfaces are well known and comparatively easy to reconstruct. The amount of light that is reflected at a given point on a surface is described by the so-called *BRDF*, the bidirectional reflection distribution function. The BRDF $f_r(x, \vec{\omega}_i, \vec{\omega}_o)$ is defined as the radiance leaving a point x in direction ω_o divided by the irradiance arriving from direction $\vec{\omega}_i$. In other words it is the directional "brightness" of a surface patch in relation to its "illumination". Although the BRDF is a flexible means of describing light interacting with surfaces it has to be noted that it cannot model all physical effects. Phosporence and fluorescence are not taken into account for example. For simplicity we also neglect the dependence on the wave length and the orientation of the surface patch (thus we assume isotropic reflection).

Then the *BRDF* can be measured easily in an experiment by examining the radiance and irradiance for a variety of incoming and outgoing angles. However, it is much more convenient to split the *BRDF* into its diffuse and specular components and examine these components separately. Most surfaces encountered in practice have diffuse (or Lambertian) reflection properties, which means that the *BRDF* does neither depend on the incoming nor the outgoing angle. A typical example is a painted wall. On the other hand the specular component of the *BRDF* is characterized by the fact that most if not all incident light is emitted in the direction of the reflection vector $r = 2(n \cdot l)n - l$ (see Figure 2.5). A mirror is an ideal

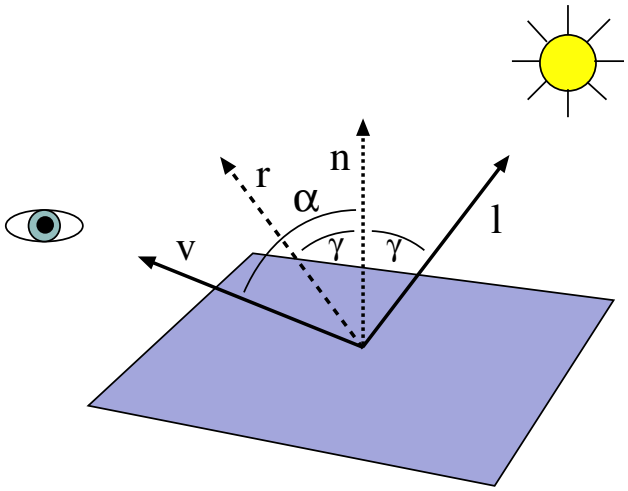


Figure 2.5: Light reflection.

specular surface, that is all light is reflected in the direction of r. Glossy materials like plastic do not have such an articulated reflection, but rather a distribution that is smeared around the reflection angle (see Figure 2.6).

In OpenGL one cannot specify the BRDF of a surface directly, but the reflection behavior can be described as a combination of ambient, diffuse and specular reflection properties. So the brightness under which a surface patch appears to the viewer is the sum of the ambient, diffuse and specular illuminations of the light sources multiplied with the respective ambient, diffuse and specular reflection terms.

The ambient reflection term is not physically valid by any means, but it is often used as a simple approximation of the indirect illumination of a scene. It is assumed that the distribution of the observed light is uniform, so that the ambient rteflection term is constant. The diffuse reflection term corresponds to the amount of light reflected off a surface in Lambertian manner meaning a uniform distribution in all directions. Since the brightness of a diffuse surface depends on the viewing angle α, the diffuse reflection term depends on the cosine of the viewing angle. The specular reflection term accounts for the specularity of a surface. The reflecivity is maximal in the reflection direction r and decreases with increasing reflection angle γ.

Let θ_a, θ_d and θ_s be the ambient, diffuse and specular reflection coefficients and let L_a, L_d and L_s be the respective illuminations. Then OpenGL calculates the brightness of a surface patch by the following formula:

$$B(l,v) = \theta_a L_a + \theta_d L_d cos\alpha + \theta_s L_s cos\gamma^{\beta} \tag{2.1}$$

The specular reflection exponent β describes the idealness of the specular reflection. An ideal mirror has a specular exponent of ∞ while plastic has a specular exponent in the range of 10-100.

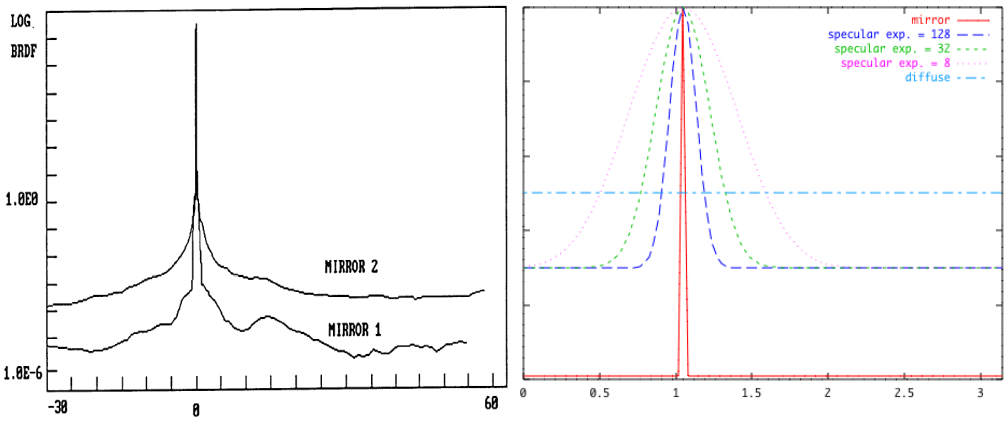


Figure 2.6: Examples of bidirectional reflection distribution functions (BRDF). **Left**: Measurement of the BRDF of two different mirrors. **Right**: BRDF of an ideal mirror and several BRDFs with different specular exponent. The green dotted BRDF is typical for plastic.

The above equation can be computed very efficiently by modern graphics hardware, so the hardware effectively carries out the simulation of a single bounce of a photon. This simplified illumination scheme is known as direct lighting or local illumination. The reflection coefficients must be determined in an experiment to fit the idealized curves with the reflection measurements. To account for anisotropic materials (velvet, brushed metal etc.) the *BRDF* also depends on the rotation angle of the surface material. Accordingly, the measurements are much more expensive. A variety of competing reflection models have been introduced for the description of anisotropic *BRDF*s, but this topic is beyond the scope of this brief introduction. The reader is referred to the PhD thesis of Wolfgang Heidrich [122] for an excellent overview on high-quality shading.

The real-time calculation of more than one bounce is currently not supported by the streaming architecture of actual graphics hardware. In order to achieve a more realistic appearance of illuminated scenes the surface colors need to be precalculated. This task is known as global illumination and can be extremely time consuming especially if refraction, diffraction or wave length dependent effects also need to be considered. Since the focus of this thesis is on real-time rendering we further restrict ourselves to the local lighting.

2.3.2 Texture Mapping

In order to achieve a realistic appearance of virtual scenes, we have seen that high-quality lighting plays an important role. Homogeneous materials like plastic can be modeled very realistically by choosing an appropriate *BRDF*. But other common real-world materials like rock, wood or carpet require additional efforts because these materials also have a specific texture. The reproduction of such materials is performed by taking a representative photo of the corresponding surface. Then this texture is ”painted” onto the surface. In computer graphics this process is known as *texture mapping*. The basic procedure is that the texture is first defined as a raster image and uploaded into the dedicated texture memory of the graphics hardware. Whenever a pixel is rasterized the hardware can now lookup the color of the pixel in the texture memory. For that purpose a mapping from world to texture coordinates is specified which uniquely defines a correspondence between each rendered triangle and the texture image.

To be more specific, each triangle vertex has a set of texture coordinates (s,t) which define the position of the vertex in texture space. These texture coordinates can be specified explicitely or implicitly by using the vertex position as texture coordinate. Afterwards the mapping is applied by multiplying the texture coordinates with a 4 by 4 matrix. Whenever a triangle is rendered the texture coordinates of each rasterized pixel are interpolated from the mapped texture coordinates of the three triangle vertices (by means of scan line interpolation). With these interpolated texture coordinates the hardware performs a lookup at the corresponding position in texture space and retrieves a color from the raster image (see also Figure 2.7). The result of this lookup determines the final color of the pixel. Optionally, the texture color can be multiplied with the shaded primary color (Equation 2.1).

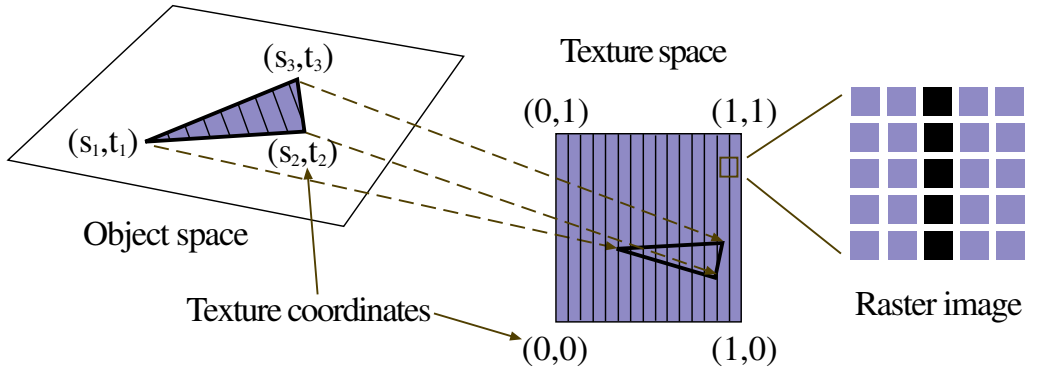


Figure 2.7: 2D texture mapping. Texture coordinates are denoted by (s,t).

The mapping from world coordinates to texture space is not an isometric projection. This means that more than one texel (that is a pixel in the texture image) can map to a single pixel on the screen. This is commonly the case when the rendered triangles are far away from the view point. In such a case the texture lookup will result in a quasi random selection of one of the texels in the footprint of the pixel. This is like shooting a bag of rice. The random colors leads to severe Moiré artifacts if the view point is moved slightly.

The Moiré artifacts can be avoided with the *MIP-mapping* technique (MIP is short for the Latin "Multum in parvo" which means many things in a small place). This technique pre-filters the texture and produces a series of shrinked images. Each shrinked image has half the size of the original one. The texture lookup now is performed in the pre-filtered image which best fits the resolution of the screen (see Figure 2.8). This is the solution for texture minification. For magnification, which means that one texel maps to several pixels on the screen, the solution is simply to bilinearly interpolate the fragment color from the four adjacent texels.

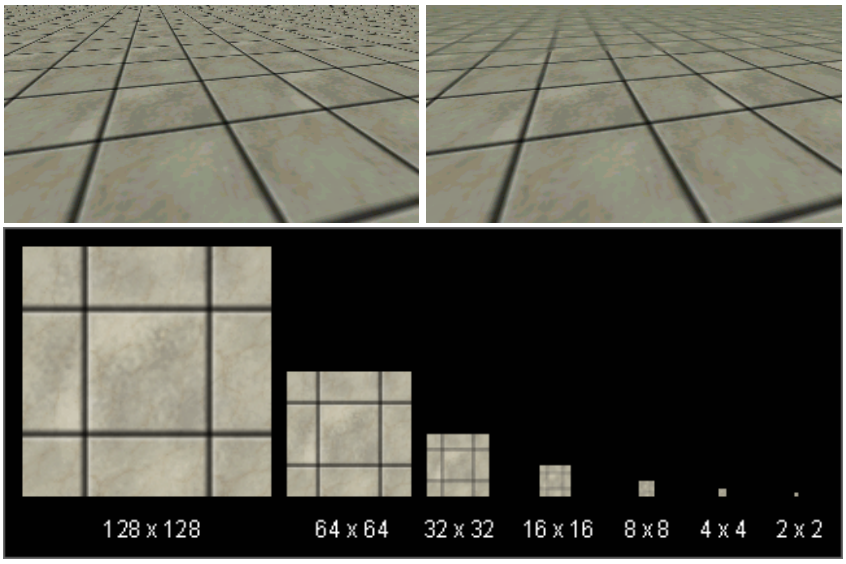


Figure 2.8: MIP-mapping. **Top left**: Without mip-mapping the Moiré pattern is clearly visible at the top of the image. **Top right**: With mip-mapping the Moiré pattern has disappeared. **Bottom**: Series of shrinked mip-map images.

Aside from pre-filtered raster images which are often also referred to as 2D mipmaps or 2D textures, the same texture mapping procedure can be also applied to three-dimensional texture data or 3D textures. In the latter case the data effectively represents a volume. This volume is addressed with a texture coordinate triple (s,t,r).

3D texture mapping is a very powerful technique that has various obvious and not so obvious application areas. One of the most obvious is the definition of a texture volume that represents the three-dimensional structure of a material. In Figure 2.9 a 3D marble texture is depicted. This marble texture has been applied to the famous Utah Teapot by also using the positions of each vertex as a 3D texture coordinate. Rather like painting an image on the surface as with 2D texturing, 3D texturing is more like sculpting an object out of a solid block of material. One of the not so obvious applications of 3D texturing, for example, is pre-integrated cell-projection as explained in Chapter 8.

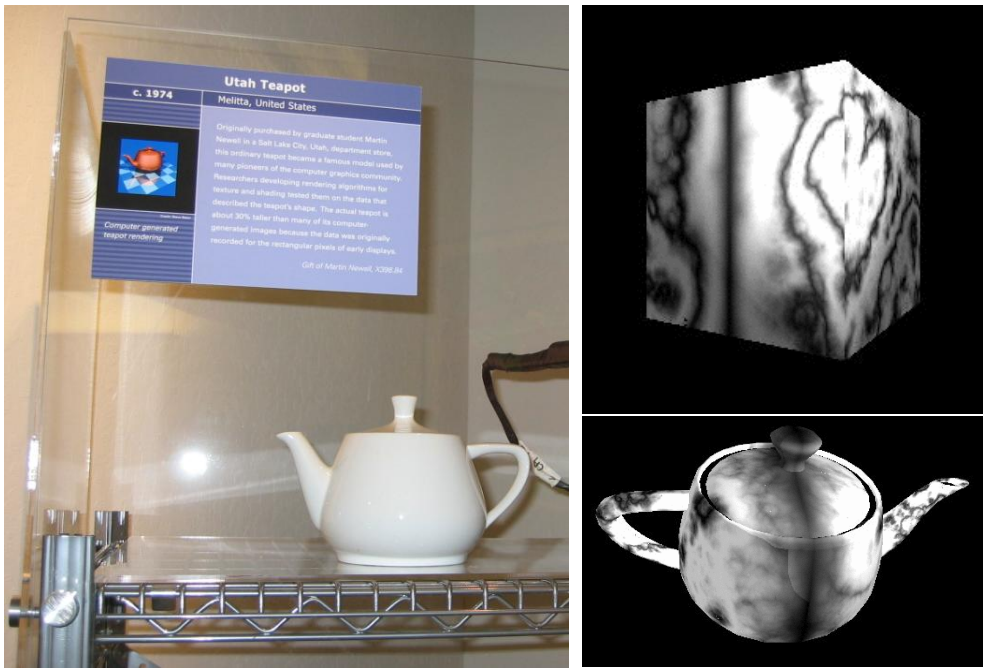


Figure 2.9: 3D-Texturing. **Left**: The real Utah Teapot of Melitta on exhibition in the Computer History Museum in Mountain View, CA. **Top Right**: Solid marble texture. **Bottom Right**: Marble texture applied to teapot.

2.4 Programmable Graphics Hardware

In this section we address the ongoing evolution of the graphics hardware. Basically, hardware accelerated rendering started with the SGI graphics workstations and Evans and Sutherland flight simulators. These graphics workstations and simulators were very expensive gadgets that were taylored to very specific visualization tasks. It was unthinkable to have one of those at home.

But in the late 90's the computer gaming industry discovered the third dimension as the key element for realistic computer games. This was a tremendous push for graphics hardware manufacturers. With the Sony Playstation launch in 1994 three-dimensional graphics for the first time was becoming cheap enough to be affordable for the broad mass.

In the subsequent years cheap powerful graphics accelerators were also becoming available for the PC platform. Here, to mention only a few, the 3dfx Voodoo and the NVIDIA TNT and GeForce products were the dominating accelerators. Only recently ATI with its Radeon products has been increasingly successful in that market and currently offers the best price performance ratio.

Before the upcoming of three-dimensional computer games, graphics hardware was organized inherently static. The graphics pipeline was a fixed function pipeline fulfilling a specific operation at each stage. For example, the perspective transformation was hardwired as the multiplication of a 4 by 4 matrix with the incoming position vector of the vertices followed by the homogeneous normalization of the vector. Lighting is performed similarly by transforming the normal vector with another matrix.

Back in the early days of computer graphics this transformation scheme was sufficient to shade the polygonal models. But current sophisticated lighting techniques do not fit as easy into the simple hardwired scheme of the original graphics pipeline. For the efficient implementation of advanced lighting and rendering techniques the pipeline had to be redesigned to be much more flexible. This was achieved by making the main stages of the pipeline customizable. In particular the vertex projection and the fragment texturing stages of the pipeline have been replaced by special purpose processing units that are customizable via low-level assembly language.

2.4.1 Vertex Shaders

The vertex shader is the customizable counterpart of the perspective transformation and the lighting operation. It uses a SIMD processing model with efficient instructions for vector multiplication, summation, dot product, and normalization. It also features control structures which direct the flow of the vector computations. In the language of the vertex shader the perspective transformation, for example, can be reformulated as four subsequent dot products and one division summing up to a total of five vertex program instructions.

The vertex program code is compiled at runtime and uploaded to the graphics hardware, which after that executes the code for each incoming vertex. The maximum program length is between 128 and 256 operations depending on the actual graphics hardware. With this restriction already quite complicated vertex programs can be written. It is expected that the maximum program length will

increase even further in the future giving way for stunning new graphics effects computed at real-time.

2.4.2 Pixel Shaders and Fragment Programs

Pixel shaders are for fragment processing what vertex shaders are for vertex processing. In comparison to a vertex program the pixel shader instructions are less powerful and program length is much more limited. Conditional jumps are only implemented in the latest graphics hardware generation, and lead to a significant slowdown of fragment processing if used too frequently. But the principle is the same as with vertex programs. This means that for each incoming fragment a pixel shader micro-program is executed which determines the final color of the fragment. The pixel shader can read various input registers, such as primary color, sampled textures, Z-values and so on to accomplish complex texturing and shading operations.

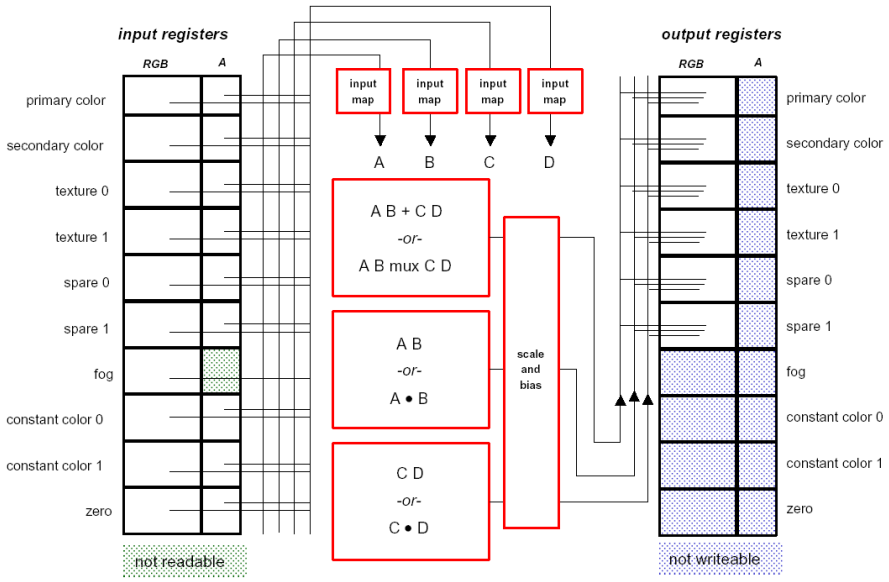


Figure 2.10: Register combiners on NVIDIA GeForce 2.

The first step towards the complex pixel shaders were the so called register combiners. They became necessary, since modern graphics accelerators allowed to define not only a single texture per fragment but up to eight textures or even more. The result of the lookup for each of the multiple textures had to be combined into one single color value which becomes the final color of the fragment. For that purpose a register combiner allows to map an input register set to an

output register set via a fixed set of simple predefined operations. By adjoining several combiner stages the range of computable functions is enlarged. The initial implementation of the register combiners in the NVIDIA GeForce offered only 2 such stages, but the GeForce3 already offered 4 stages (see Figure 2.10) and more powerful operations. The logical consequence in the development of the fragment processing unit was not just to add more and more register combiner stages but to introduce a fully programmable unit, which is now known as the **fragment program** in OpenGL notation or the **pixel shader** in DirectX notation.

To give an example of a fragment program the following code samples a texture in the same fashion as the original uncustomizable pipeline, that is the following program shows how to perform a simple modulation between the interpolated primary color and a single texture lookup as done by the default texture environment of OpenGL:

```
!!ARBfp1.0

ATTRIB tex = fragment.texcoord;      # texture coordinates
ATTRIB col = fragment.color.primary; # interpolated color
OUTPUT outColor = result.color;
TEMP tmp;

TXP tmp, tex, texture, 2D;           # sample the texture
MUL outColor, tmp, col;              # perform the modulation

END
```

For completeness the DirectX version (without modulation) is given below which differs mainly in the declaration syntax but otherwise the pixel shader offers almost the same instruction set and functionality as a fragment program:

```
ps_2_0

; Declare the s0 register to be the sampler for stage 0.
dcl_2d s0
; Declare t0 to have 2D texture coordinates from stage 0.
dcl t0.xy

; Sample the texture at stage 0 into register r1.
texld r1, t0, s0
; Move r1 to the output register.
mov oC0, r1
```

After the final color of a fragment has been computed, the last step is to blend the color of the fragment with the color in the frame buffer. This is necessary

to implement transparency. As opposed to vertex and fragment processing, the blending stage of the pipeline has remained comparatively unchanged since the early days of computer graphics. The blending stage offers a fixed set of operations to mix the incoming color with the color in the frame buffer. In our opinion it is only a matter of time when this stage also becomes freely programmable (e.g. by means of a blending combiner).

Chapter 3
A Brief History of Terrain Rendering

The first prerequisite in enriching outdoor scenes with natural volumetric effects is to have basic knowledge about the fundamental principles of outdoor rendering and in particular the key component, the terrain renderer. In this chapter we therefore give a brief chronological survey of existing terrain rendering algorithms and try to predict the influence of future graphics hardware development.

3.1 Data Representation

Traditionally, terrain data is available mostly as contour map. During the last century geographers have been collecting contour maps for nearly every point on earth. Nowadays data acquisition via remote sensing, that is via airborne or satellite based scanners, plays a more and more important role. In this context, topographic information is represented by a so-called height field, that is a regular height matrix. The matrix is often given as a grey-scale or color-coded image where the brightness or color of each pixel correlates to its elevation (for an example see middle image in Figure 3.1). The actual size of a height field can range from 1000 by 1000 grid points to any arbitrarily large number. The whole earth at a resolution of 1km, for example, corresponds to roughly $\frac{1}{2}$ billion data points. Due to the huge amount of data and the large size of the corresponding triangle mesh, height fields usually cannot be rendered exactly. Instead, a simplified mesh is produced and displayed. As we will see in Chapters 10 and 11, the same simplification strategy is suitable for volume rendering of natural gaseous phenomena.

3.2 TINs

The so-called triangular irregular networks (commonly abbreviated by the term TIN [30]) were the first algorithmic attempts to implement a terrain simplification strategy (Military flight simulators have been using simplification schemes since the seventies but little is known about their algorithmic strategy). The regular

height field is converted into a irregular triangle mesh which contains less triangles in regions that are smooth and more triangles in regions which have high surface curvature. Due to the irregular structure the achieved compression ratios are high. A complete mountain ridge or a flat riverbed, for instance, can be represented with just a few appropriately shaped triangles (compare Figure 3.1).

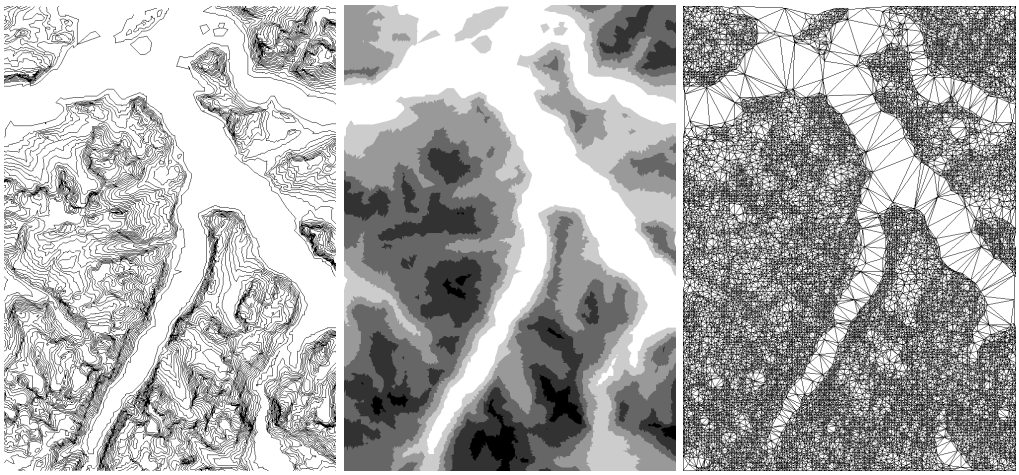


Figure 3.1: Example view of a *T*riangular *I*rregular *N*etwork (TIN). First a contour map (left) is converted into a height field (middle). The brightness of each grid point corresponds to its elevation. Then the height field is tessellated in such a way that low surface curvature leads to larger triangles (right) as can be seen at the river beds.

There exist a variety of different algorithms which try to squeeze the size of the irregular mesh while preserving the structure of the original data (including wavelet based encodings [33]). However, the basic principle is the same for all these algorithms, so we do not give a detailed analysis here. It is more important to point out that these approaches have one fundamental disadvantage: Depending on the compression ratio small details of the landscape are smoothed out during simplification. If the viewer is far away from these details, one might not notice their absence, but in the near vicinity the absence becomes observable. For this reason, TINs are not suitable for a variety of application areas such as low level flight simulations or proximity warning for civil aviation. In the next sections we outline view-dependent terrain rendering algorithms which circumvent the mentioned disadvantage.

3.3 S-LOD

As mentioned in the previous section, the huge amount of data prevents the terrain model from being displayed in full detail. Put in another way, the large extent of a terrain leads to a projective size of much less than one pixel for distant details (compare Figure 3.2). While it would be overkill to draw all such detail, it still must show up if the viewer is close enough. This kind of view-dependency was the main problem which had to be dealt with in the early days of terrain rendering.

One of the first solutions to this problem was the so-called static level of detail (S-LOD [50, 107]) technique. Here the terrain is divided into tiles each of which is represented by a set of TINs with varying resolutions. Depending on the distance to the viewer for each tile a TIN with appropriate projective triangle size is chosen from the set. If regularly coarsened meshes are used instead of TINs the method is called geo-mipmapping [15]. The appropriate resolution of each tile is chosen in such a way that the projected screen space error of each individual tile is just below a predefined error threshold of one or several pixels. If the error is below one pixel the simplified terrain is indistinguishable from the original mesh. Special care has to be taken to close the gaps that may arise in between the tiles.

In practice, much higher errors than one pixel have to be admitted to achieve sufficient frame rates. Since the estimation of n of the screen space error of an entire block is very conservative a lot of small redundant triangles are generated if the block contains only just a few small details. So a block-based triangulation is only a very rough approximation to the optimal triangulation. As a consequence, far more triangles are rendered than necessary and the frame rates achieved for an error threshold of one pixel are just not sufficient. Therefore higher thresholds are usually allowed, but then the transition of one level of detail to another becomes visible. This temporal artifact is known as the popping effect. As the human eye is very sensitive to sudden temporal changes these artifacts lead to serious distraction of the observer and should be avoided whenever possible. Another problem that arises from tiles with different resolutions is the issue of building a conforming mesh. In order to avoid gaps between adjacent tiles of different resolution, additional triangles are needed to connect the tiles properly.

3.4 Progressive Meshes

A special case of the S-LOD technique are progressive meshes as proposed by Hoppe [42]. While the progressive meshes technique was originally designed for the incremental simplification and transmission of three-dimensional objects the technique can also be applied to terrain rendering. Each tile of the terrain is represented by a set of meshes which are derived from the original mesh by the well

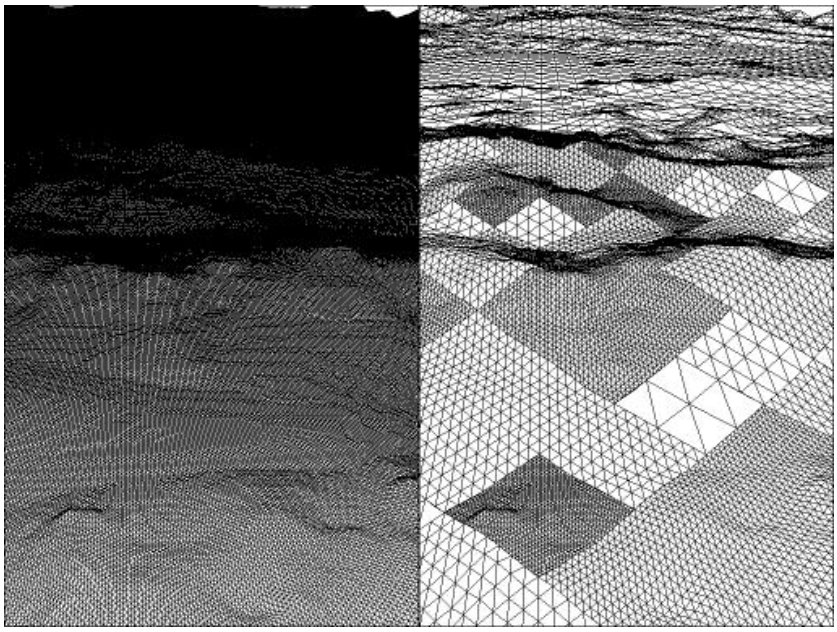


Figure 3.2: Static LOD as proposed by Koller et al. (Images taken from [50]): On the left side the original mesh is shown, whereas on the right side the tiled representation is depicted. For each tile a set of meshes with increasing coarseness is built. Depending on the distance to the viewer, the mesh is selected which has a projective screen space error that does not exceed a certain predefined limit. This limit is usually set to one or several pixels. If the limit is below one pixel the simplified mesh is indistinguishable from the original mesh.

known split and merge operations of progressive meshes (see [40, 41] for detailed information). The connectivity of the tiles is ensured by a full grid at the borders. Figure 3.3 shows an example in which different tiles are depicted by alternating colors. The advantage of this approach is the good approximation quality but on the downside the memory requirements are huge and complex data structures have to be maintained. In summary, progressive meshes are a good choice for the simplification of arbitrary 3D geometry but for the special case of height fields there exist specialized algorithms which are much more straightforward and easier to implement as we will see in the following.

3.5 C-LOD Algorithms

The most elaborate terrain rendering technique known today is the continuous level of detail technique (C-LOD). It improves the sub-optimal approximation

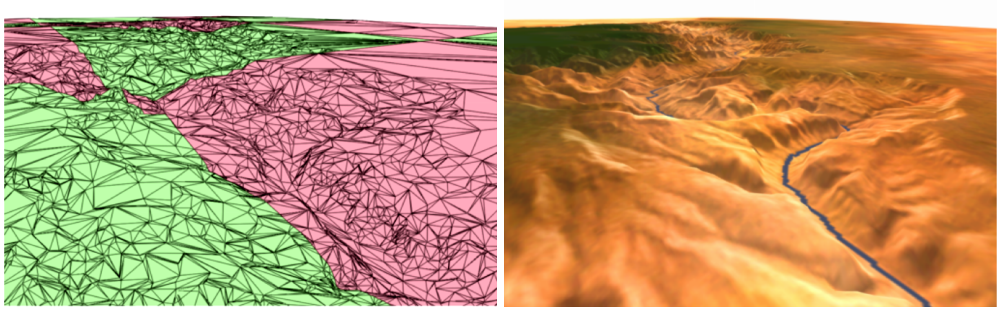


Figure 3.3: Progressive meshes as proposed by Hoppe (Images taken from [42]). Different tiles are depicted by alternating colors.

quality of the S-LOD algorithms in a sense that the triangulation is altered on a per triangle and not on a per tile basis. This allows much better approximations which adapt optimally both to the viewing distance and to surface roughness. If, for example, a tile has a single small peak, the S-LOD algorithm needs to choose a high resolution mesh for the entire tile (compare dark tile at the bottom left of Figure 3.2). The C-LOD method does not exhibit this restriction, since the triangle count can be increased for the small peak only. In the following sections the three main C-LOD algorithms by Lindstrom [59], Duchaineau [19], and Roettger [89] are discussed in chronological order.

3.5.1 Lindstrom's Algorithm

The first published C-LOD algorithm which achieved consistent interactive frames rates and high image quality is the approach of Lindstrom et al. [59]. It applies a two-step simplification scheme, that is a block- and a vertex-based simplification step. Both steps are driven by the screen space approximation error of the mesh. A coarse level of simplification is performed to select discrete levels of detail for blocks of the surface mesh, followed by further simplification through repolygonalization in which individual mesh vertices are considered for removal. These steps compute the appropriate level of detail dynamically in real time, minimizing the number of rendered polygons and allowing for smooth changes of resolution across areas of the surface (see Figure 3.4).

The conditions under which a triangle pair can be combined into a single triangle are primarily described by the amount of change in slope between the two triangles. The maximum vertical distance between the two configurations induced by the omission of one vertex is referred to as the delta value δ of each vertex. As the delta value increases, the chance of triangle fusion decreases. By project-

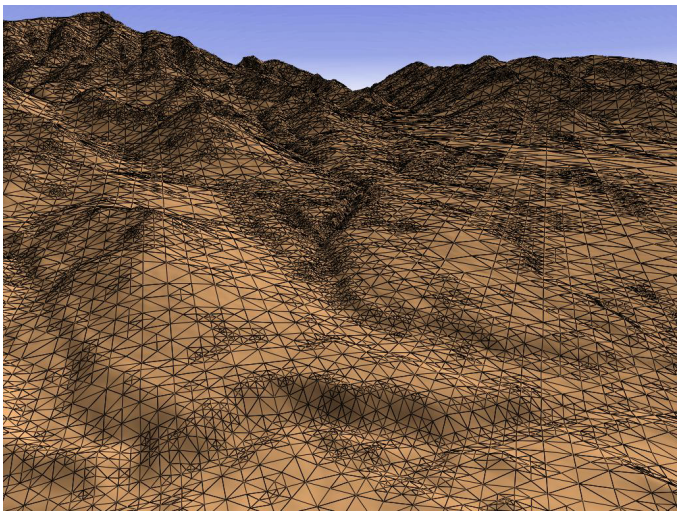


Figure 3.4: C-LOD algorithm as proposed by Lindstrom et al. The wire frame view shows a triangulation with approximately 40,000 polygons (Images taken from [59]).

ing the delta segment onto the projection plane, one can determine the maximum perceived geometric error between the merged triangle and its corresponding sub-triangles. If this error is smaller than a given threshold τ, the triangles may be fused to reduce the complexity of the surface mesh. If the resulting triangle has a co-triangle with error smaller than the threshold, this pair is also considered for simplification. This process is applied recursively until no further simplification of the mesh can be made.

Let $\vec{e}$ be the view point and let $\vec{v}$ be the position of each vertex. Furthermore, let d be the distance from $\vec{e}$ to the projection plane and define λ to be the number of pixels per world coordinate unit in the screen coordinate system. With these definitions, the length of the projected delta segment as shown in Figure 3.5 is approximated by the following equation:

$$\delta_{screen} = \frac{d\lambda\delta\sqrt{1-\left(\frac{\vec{e}_z-\vec{v}_z}{||\vec{e}-\vec{v}||}\right)^2}}{||\vec{e}-\vec{v}||} \tag{3.1}$$

The square root term equals the sine of the viewing angle and accounts for the vanishing error at nadir viewing angles (that is looking down from straight above). The denominator accounts for decreasing error related to increasing distance.

In principle, the length of the projected delta segment needs to be computed for every triangle pair in order to check whether it is fused or not. But complex

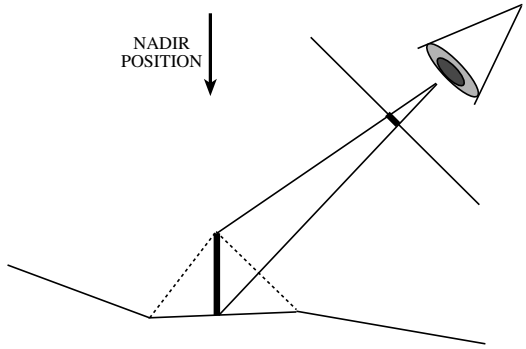


Figure 3.5: The projection of a delta segment onto the viewing plane. The delta segment and its respective projection are depicted by bold lines. For nadir view positions the projected length vanishes. For larger distances of the segment to the view position the projected length also becomes smaller and smaller.

datasets may consist of millions of polygons, and it is clearly too computationally expensive to run the described simplification process on all polygon vertices for each individual frame. By obtaining a conservative estimate of whether certain groups of vertices can be eliminated in a block, the mesh can be decimated with little computational cost. If it is known that the maximum delta projection of all lowest level vertices in a block falls within τ, those vertices can immediately be discarded, and the block can be replaced with a lower resolution block, which in turn is considered for further simplification. Accordingly, a large fraction of the costly delta projections can be avoided.

To efficiently render the mesh, a graphics primitive such as the tri-stripping primitive supported by OpenGL may be used. For each specified vertex *v*, the previous two vertices and *v* form the next triangle in the mesh. At certain points, the previous two vertices must be swapped via an additional `glVertex()` call, but basically a complete block can be rendered with a single graphics primitive.

The most prominent advantage of the described C-LOD algorithm is the possibility to maintain a desired quality of the triangulation via the screen space error threshold τ. Due to the view-dependent triangulation small distant details need not be represented with the same number of triangles than those which are nearby. This leads to a tremendous reduction of the number of rendered polygons. On the down side the triangulation has to be updated for each frame, which leads to heavy CPU utilization. Especially the block switches are very costly and may cause frame drops on slower platforms. Since the triangulation is different for almost every frame, vertex arrays and other related performance optimizations are difficult to apply. In many application scenarios it is desired to guarantee a maxi-

mum triangle count. Although the number of generated triangles correlates tightly with the error threshold it is not possible to guarantee a maximum triangle count directly. Negative feedback must be used to steer the triangle count by smoothly adapting the error threshold.

3.5.2 Duchaineau's Algorithm

Following the Lindstrom paper, a major improvement of the C-LOD technique was achieved by Duchaineau et al. [19] (see also Figure 3.6). They presented an algorithm with optimized error metrices and guaranteed error bounds that achieves specified triangle counts directly and uses frame-to-frame coherence to operate at high frame rates. The method was dubbed Real Time Optimally Adapting Meshes (ROAM). It uses two priority queues to drive split and merge operations that maintain continuous triangulations built from preprocessed bintree triangles. ROAM execution time is directly proportional to the number of triangle changes per frame, hence performance is almost insensitive to the resolution and the extent of the input terrain. Just as the square-shaped quadtree has a triangle-quadtree counterpart, the familiar rectangle shaped bintree [94] has a little-known triangle-shaped counterpart. The children of the root are defined by splitting the root along an edge formed from its apex vertex to the midpoint of its base edge. The rest of the bintree is defined by recursively repeating this splitting process. A key fact about bintree triangulations is that neighbors are either from the same bintree level or from the next finer level for left and right neighbors, or from the next coarser level for base neighbors. A simple split operation and its inverse are depicted in Figure 3.7 for a triangulation containing a so-called diamond.

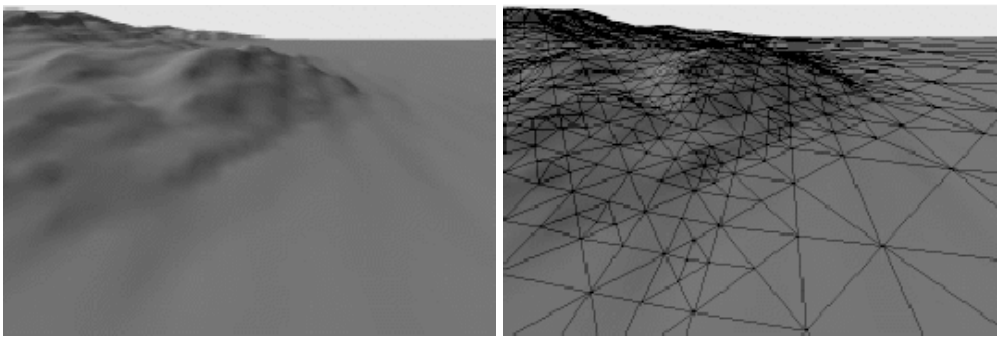


Figure 3.6: C-LOD as proposed by Duchaineau et al. (Images taken from [19]): Example of ROAM terrain.

An important fact about the split and merge operations is that any triangulation may be obtained from any other triangulation by a sequence of splits and merges.

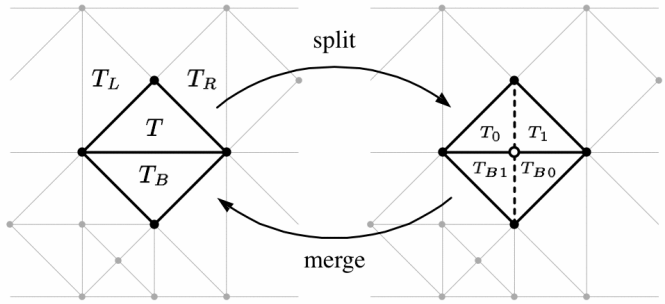


Figure 3.7: Split and merge operation on a diamond (from [19]).

The split and merge operations provide a flexible framework for making fine-grained updates to a triangulation. No special efforts are needed to avoid cracks or T-vertices. The idea of a greedy algorithm that drives the split and merge process is simple: keep priorities for every triangle in the triangulation, and repeatedly do a forced split of the highest-priority triangle. This process creates a sequence of triangulations that minimize the maximum priority. The only requirement to ensure this optimality is that priorities are monotonic, meaning that a child's priority is not larger that its parent's. This is a valid assumption, since the priorities typically correlate to a monotonic error bound. Adding a second priority queue for mergable diamonds allows the greedy algorithm to start from a previous optimal triangulation when the priorities have changed, and thus take advantage of frame-to-frame coherence.

The basic error metric of ROAM is the distance between where each surface point should be in screen space and where the triangulation places the point. Over the whole image the maximum of these pointwise distortions is measured. In this sense Duchaineau's approach is very similar to Lindstrom's. Besides this basic error metric the priority-driven mesh generation allows further advanced error metrices:

- **Back-face detail reduction**: Priorities can be set to a minimum for triangles whose subtree of triangles are all back-facing.
- **Normal distortion**: Priorities should be increased at vertices with large normal distortion to reduce specular high-lighting artifacts.
- **Texture coordinate distortion**: Priorities should also correlate to texture distortion.
- **Silhouette edges**: Specific emphasis can be placed on triangles whose normal bounds indicate potential back-face to front-face transitions.

- **View frustum**: Priorities outside the viewing frustum can be set to a minimum.
- **Atmospheric obscurance**: Where fog reduces visibility, priorities can be reduced.
- **Object positioning**: To correctly position objects on a terrain, the priorities of triangles under each object can be artificially increased.

The screen-distortion priorities of the triangles change as the viewing position changes, typically in a slow and smooth manner. Recalculating priorities of all triangles for every frame is too costly, especially for some of the advanced error metrices. Instead, priorities are recomputed only when they potentially affect a split/merge decision. Recomputation of a triangle can safely be deferred until its priority bound overlaps the crossover priority. A deferral list is kept for each of the next few dozen frames. Only the triangles on the current frame's deferral list must have priorities recomputed. If time allows, additional triangles may be recomputed in subsequent deferral lists. The total memory requirements of ROAM range from 8 to 20 bytes per vertex depending on the specific implementation. To give a practical example, the memory footprint for a 2000 by 2000 height field ranges from 31 to 76 MB.

The main advantages of ROAM are its flexibility with respect to applicable error metrices and the guaranteed triangle count which can be achieved with low computational overhead. But even the implementation of only a subset of the proposed additional error metrics is a very complicated task. For a specific type of application one needs to know in advance whether the described sophisticated features are demanded or if a much simpler approach is sufficient. For example, the excessive use of linked data structures, such as the deferral lists, is not preferable in interactive entertainment, since main memory can become heavily fragmented after long periods of game play. Here a much simpler approach can lead to a much more stable algorithm which is also much easier to implement and verify.

3.5.3 Roettger's Algorithm

Albert Einstein: *Everything should be as simple as it is, but not simpler!*

In the last section we have described the development of terrain rendering algorithms over the years. While these algorithms are already quite mature, the demands of interactive entertainment are quite different from those encountered in the academic arena: Simplicity, stability, and time-to-market are often much

underrated. An example of an algorithm that meets the requirements of interactive entertainment is the simple yet efficient C-LOD algorithm presented in [89]. We have been developing this algorithm in cooperation with Massive Development in 1997/98. It is included in the Krass game engine which is the graphics core of the multi-award winning computer game AquaNox and its successor AquaNox:Revelation. The terrain renderer features low memory consumption and efficient geomorphing. As we will see these features are very important for outdoor game engines.

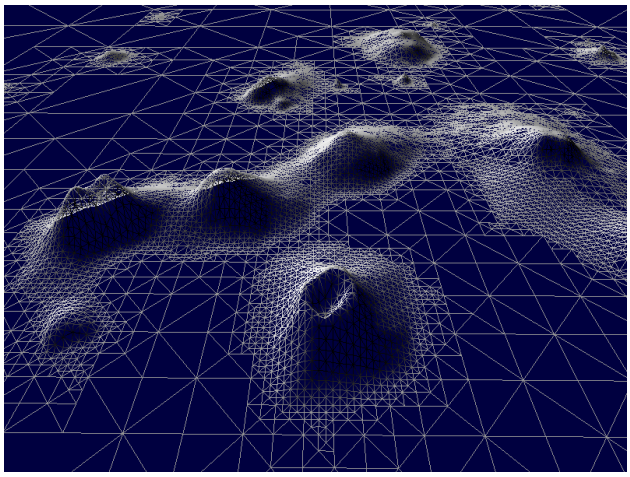


Figure 3.8: C-LOD as proposed by Roettger et al.: Wire frame view of the Galapagos Islands.

The algorithm is based on a quadtree representation of the height field, which is stored as a compact quadtree matrix. Besides the elevation data only this matrix needs to be stored in main memory. This results in a memory footprint of either 3 or 5 bytes per heixel (as an analogue to a pixel a heixel is a height field element) for 16 bit or floating point height values, respectively. In Figure 3.8 an example triangulation of the Galapagos Islands is shown. The corresponding schematic view of the quadtree is given in Figure 3.9.

Each node of the quadtree corresponds to a maximum of 8 triangles organized as a triangle fan around the node's midpoint as shown in Figure 3.9 (also compare left middle case in Figure 2.3). A conforming mesh is obtained simply by skipping those vertices of a triangle fan which are a T-vertex. In Figure 3.9 the skipped vertices are depicted by crosses. The triangulation for a specific point of view is calculated by evaluating a decision criterion for each quadtree node in a top-down fashion. Being more specific, the criterion is first evaluated for the root node. If the criterion is true for the root node its four children are checked using depth-

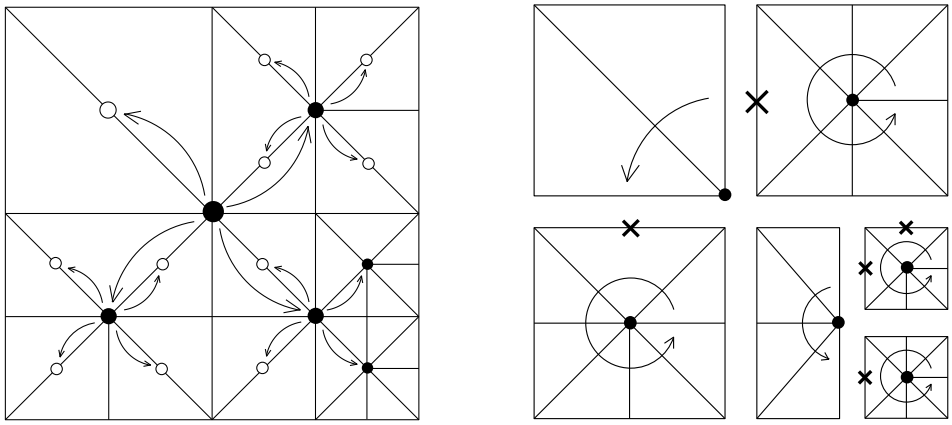


Figure 3.9: **Left**: Sample triangulation of a 9 by 9 height field. Parent-child relations of the quadtree are indicated by arrows. **Right**: The same triangulated height field decomposed into triangle fans (depicted by circular arrows). A conforming mesh is built by simply skipping those vertices that are marked with crosses.

first traversal. This process is continued recursively until a specific node does not fulfill the criterion which means that this node is a leave node of the quadtree. Whether a quadtree node is refined or not is calculated by the following rather simple criterion:

$$f = \frac{l}{d \cdot C \cdot \max(c \cdot d2, 1)} \ , \ \texttt{subdivide if } f < 1 \tag{3.2}$$

Here, l denotes the distance of the viewer to the midpoint, d denotes the edge length, and $d2$ defines the local surface roughness of each node. As a result, the level of mesh refinement is determined by the distance to the viewer and the local surface curvature as defined by the precomputed $d2$-values. The constants c and C determine the global and minimum resolution of the resulting triangulation. The total number of generated triangles is closely coupled to the global resolution parameter c, so that on fast machines this parameter can be chosen to be larger than on slow machines to obtain a finer triangulation of the terrain.

The given formula computes an approximation of the projection error of each node. The criterion is a reformulation of Lindstrom's delta segment error (see Figure 3.5). For simplicity the square root term in Lindstrom's formula has been neglected. This means that the refinement does not depend on the viewing angle, since this would introduce significant computational overhead as illustrated by Lindstrom's approach. If one only considers geometric aberration, then angular dependency makes sense, but if one also considers the influence of texturing and lighting then angular dependency makes less sense. This statement is illustrated

in the following example: Consider looking onto a flat plane with a small peak in birds eye view. Then the geometric error induced by the peak is very small since the viewing angle is nearly 90 degrees. So the triangles of the peak probably will be fused and the peak will be flattened out. Now, if you take per-vertex lighting into account the peak should still be visible because of the change of the surface normal. But since the peak is flattened out we also loose the surface normals which means that the peak is not shaded correctly. Due to perspective distortion it also makes a difference whether we render a textured peak or a flat surface with the same texture. In summary, angular dependency is not compatible with per-vertex lighting and texturing. For this reason we decided not to include angular dependency in our approach. This keeps the decision criterion simple which in turn also results in a very fast, robust, and simple algorithm consuming only a minimum of extra memory per grid point.

Since C-LOD approaches generate view-dependent triangulations, the so-called popping effect leads to a serious distraction of the observer. When approaching a surface detail from the far distance the surface detail will suddenly pop up at a specific point. If the screen space error is below one pixel this popping effect is not visible but in many situations one cannot afford the resulting high triangle count. In these cases a technique called geomorphing eliminates the popping effect: Instead of letting the surface detail pop up suddenly, it is blended in smoothly. For this purpose the elevations of the vertices of a quadtree node which is marked for refinement are smoothly interpolated between the corresponding two quadtree levels. In contrast to a sudden pop, a smooth interpolation is hardly noticable by a human viewer.

Our terrain rendering algorithm is especially tailored to efficient geomorphing. The interpolation of the vertices is not carried out in a fixed time interval. Instead, the speed of morphing is coupled to the screen space error which yields a much better suppression of the popping effect. This geomorphing scheme reliably prevents the popping effect up to an screen space error of approximately 10 pixels. On a Linux PC equipped with an AMD Athlon with 1.2 GHz and an NVIDIA GeForce3 we achieve about 105,000 geomorphed, textured, and lit triangles per frame at a target frame rate of 30 Hertz. This corresponds to about 3 million vertices per second.

Back in 1996, when the C-LOD algorithm was presented, a threshold of $\tau = 3$ resulted in a frame rate of approximately 30-50 frames per second. But then the popping effect was already observable. Nowadays the graphics performance has increased significantly, and it is no longer a problem to maintain a screen space error of less than one pixel at frame rates above 50 Hertz. As explained in Chapter 4 the situation is different in interactive entertainment, where many tasks are carried out concurrently and there is less time available for terrain rendering. In such a setting geomorphing is still required.

3.6 Future Development

With respect to the future development of terrain rendering algorithms, one can foresee a main development branch. Due to increasing programmability the graphics hardware is taking over many tasks that previously had to be carried out by the CPU. With respect to terrain rendering, for example, the Matrox Parhelia is able to interpret a 2D texture map as a height field. It generates the surface triangles on chip to minimize bus traffic. This procedure has the drawback that the size of a height field is limited as it has to reside in texture memory. Larger terrains require a hierarchy similar to the S-LOD technique. For this reason the S-LOD technique will celebrate its come back in the near future as indicated by recent publications [8], but the problem of maintaining a conforming mesh is yet to be solved on the graphics hardware side. Aside from any uncertainties of future development one thing is almost certain: More and more hardware-accelerated alternatives to traditional terrain rendering algorithms will emerge.

Chapter 4
The Terrain Rendering Pipeline

Figure 4.1: An example screen shot of the computer game AquaNox showing volumetric effects like fog and a jet-wash.

With the upcoming of advanced terrain rendering algorithms (see Chapter 3) the complexity of the outdoor scenes displayed in interactive entertainment has increased significantly over the past years. By today's standards the most efficient method to display a terrain is the continuous level of detail technique (C-LOD; see Section 3.5). Despite its advantages these techniques are nowadays just beginning to migrate into the design of modern 3D graphics engines. With the demand for more and more complex outdoor scenes this situation will clearly change in the future.

With the C-LOD algorithm being the key component for the real time display of large outdoor scenes, there exist a variety of other aspects that have to be considered to achieve the desired look and feel of an organic landscape. From a game developers point of view, terrain rendering is a data driven process, which not only involves the real time display of a given terrain but also the design of the artificial landscapes and the realistic texturing and lighting thereof. The entire story can

be described as what is called the terrain rendering pipeline. Following the data flow from the beginning to the end of the pipeline, in this chapter we describe the terrain renderer as implemented by the Krass game engine of Massive Development. For the modeling of the surface properties we introduce the three functional groups illumination, material, and global effects. On the one hand, this separation offers high flexibility with respect to the visual appearance of the surface. On the other hand, each functional group results in the application of one or more surface textures which can be rendered efficiently using multi-texturing. In addition, the rendering process can be easily divided into several distinct passes, which makes it possible to customize the entire pipeline for different graphics hardware. This approach has been successfully demonstrated in the DX8 computer game AquaNox [87].

The terrain rendering pipeline consists of 6 main stages which are described in the following (see also Figure 4.2):

4.1 Landscape Data Generation

The generation of terrain data is a complex task, which is often underestimated. The terrain data has to satisfy several requirements. Obviously, the visual appearance is of prime importance. Furthermore, the topology of the terrain is one of the key elements for the subsequent mission design process. Last but not least, the data generation process should be cheap in terms of time and money. As a first step, real world terrain data is collected. This data source guarantees the natural appearance and authenticity. For the purpose of mission design, the level designer uses common image editing tools. These tools are utilized to manually generate displacement maps, containing the features which are relevant for the game play. In a final step, the height field is filtered in various ways (noise, edge enhancement, etc.). To avoid quantization artifacts the entire process is carried out with at least 16 bits of accuracy.

4.2 Real Time Display of the Terrain

Once the terrain is defined by a two-dimensional cartesian height field, sophisticated algorithms are needed to display the landscape in real time. The size of a height field easily exceeds 1024x1024 grid points, which in turn corresponds to more than 2 million triangles that have to be rendered in each individual frame. Since the exact display is not feasible, the Krass game engine applies the current state of the art in this area, that is the C-LOD technique [89] as described in the previous chapter.

In interactive entertainment, however, terrain rendering is only one task among many others that have to be carried out for each frame. Therefore the affordable screen space error targeted by the terrain renderer is usually well above one pixel. Larger screen space errors manifest themselves in the popping effect. As a solution to this problem, the geomorphing technique smoothly interpolates between the different levels of detail effectively rendering the popping effect invisible [12, 89]. This allows smooth immersive terrain visualizations even on low end graphics hardware. Since the morphing operation needs to be carried out at least every 100 milliseconds to keep the illusion of a static triangulation, frame to frame coherence is difficult to exploit. In order to speed up terrain rendering, the view frustum is predicted for the next 100 milliseconds. Then the triangulation is computed for the predicted and enlarged visible area. Until the next update of the triangulation the generated triangles are cached. In this way smooth visualizations of a terrain are generated at real time and with a low average CPU load. This concept is called semi-dynamic terrain generation.

4.3 Terrain Material

The terrain material is assembled from three textures, a coarse color texture, a finer material texture, and an even finer detail texture. It is important to differentiate between color and structure. The coarse color map is used to generate the large scale coloring of the entire terrain. The material map represents the structure of a material at a mid-frequency level, whereas the detail map contains only intensities at a high-frequency level. The latter map is used to represent small details close to the viewer.

4.4 Terrain Illumination

The illumination is constructed by summing up the emission of all static and dynamic light sources. The static light map covers the entire terrain and is generated in a preprocessing step, which gathers the ambient and diffuse contributions of all static light sources. Since a simple ray casting strategy is used to calculate the incoming intensity for each texel of the light map, static shadows are already included at this point. As the terrain geometry is generated by a C-LOD system, it only makes sense to perform the dynamic lighting calculation on a per-pixel basis. For this purpose, a dynamic light map is generated, which only covers the view frustum in order to ensure a sufficient texture resolution. Using the light map as a render target, the incoming light is accumulated for each dynamic light source. The light intensity is calculated by applying a dot product between the normal

map of the terrain and a radial light field which is specific for each light source. At this point, dynamic shadows are also taken into account. A bounding sphere approximation of each dynamic object is projected onto the terrain to maintain a shadow buffer in the alpha channel of the render target. The alpha channel represents the height of the object. When rendering each dynamic light source, we have to determine whether each texel is shadowed or not. This is achieved by comparing the height of the light source with the height coded in the alpha channel. This concept is applied to all dynamic lights including caustics and the sun light. A total of more than 500 light sources can be treated in real time using this texture based lighting approach.

4.5 Organic Features

In the next step organic features are added to the scene. We distinguish between local and global phenomena. As an example for local phenomena a large scale plant rendering system is used. The plants are categorized into several groups. For each group a density distribution is painted by the level designer. According to this distribution, the plants are placed in a pseudo-random fashion. The plant seeding is performed on the fly for the visible part of the terrain only. In order to maximize the geometry throughput, the plants are cached based on a tiling scheme. As an example for a global phenomenon a particle system is used for the display of floating plankton. This particle system has been designed to run entirely in the vertex shader of DX 8 graphics hardware.

4.6 Global Volumetric Effects

The last stage of the pipeline handles volumetric effects such as fog or water turbidity. In general, the visualization of volumetric effects requires the solution of the light transport equation. The Krass game engine uses a special type of volumetric fog which is defined to occur below a specific base height. In this restricted case, the light transport equation simplifies to a two dimensional ray integral for a constant height of the viewer. The solution of this integral is pre-calculated and stored in a standard 2D texture which is mapped over the entire scene [57, 39] (see also Section 5.3). This method avoids the appearance of popping artifacts, since the fog attenuation and emission is calculated on a per-pixel basis. Interestingly enough this feature is not only a visual effect but also offers game play relevant elements such as hiding in misty valleys.

In summary the stages of the terrain rendering pipeline are depicted in Figure 4.2.

4.7 Volumetric Effects in Practice

The question is now the following: What volumetric effects are used in practice apart from the mentioned simple fog model? The answer reduces to a mere two words: Not many! Billboard techniques (see Section 5.5) are utilized for the display of explosions, jet-washs, and smoke for instance. The latter are utilized mainly because they have low algorithmic complexity and do not require a large rendering overhead. Since there is no need for disproportionate realism in computer games a lot of eye-candy can be realized even with such a simple strategy. On the other hand more flexible fog models would offer more degrees of freedom with respect to level design.

Another volumetric effect which is often encountered in computer games is the so called sky dome (also see Section 5.1). The sky dome is a hemisphere which is textured with the photograph of a real sky. While this allows a nice looking sky at merely no cost one cannot use this approach in a flight simulator, for example, since it is not allowed to fly into the clouds.

The direction of future development clearly is to allow arbitrary view points, and in particular view points in the cloud layer. This can only be achieved with truly volumetric fog and cloud representations.

Figure 4.2: The main stages of the terrain rendering pipeline in the AquaNox game engine.

Chapter 5
Natural Gaseous Phenomena

Mr. Horse: ... *No Sir, I don't like it!*

In general, the display of volumetric gaseous phenomena is a non-trivial task. Without going into detail here, solutions to this problems exist for a variety of cases where additional assumptions reduce the complexity of the problem. The general problem, however, is closely related to volume rendering which is still a very active research area. The inherent complexity of volume data makes it very hard to design an algorithm that displays large volume data sets in a reasonable time or even worse at real time. Medium sized data sets, that fit into the dedicated texture memory of consumer graphics hardware, can be displayed using the slicing method (compare [26]). But for large volumes current volume rendering approaches achieve interactivity only by massive parallelization [53] or the utilization of special purpose graphics hardware [83, 72]. A general introduction into the topic of volume rendering is given in Chapter 6. For the purpose of the real time display of gaseous phenomena specially taylored algorithms are presented in Chapter 10 and 11. For now, we resort to those special cases which can be handled easily due to additional constraints on the volume data.

If the medium through which a light ray travels is assumed to have a constant density, the absorption along each ray of light can be expressed in terms of a simple formula. In each small step the light travels a certain fraction of its intensity is absorbed by the medium. This means that the absorbed light is transfered into thermal energy. The energy may not be absorbed completely, but usually a certain amount is also scattered and sent out again into other ray directions. The back and forth scattering of the light is one of the main reasons why volume rendering is computationally very expensive. Thus many approaches try to neglect scattering which leads to the formulation of the ray integral as described in Chapter 6.

If the direction of the scattered light is evenly distributed, one speaks of isotropic scattering. Otherwise one speaks of anisotropic scattering. The scattered light may be of the same wave length, but this is no necessity. For example, the effect known as Rayleigh scattering means that the intensity distribution of the scattered light depends on the frequency of the light. This effect is responsible for the blue sky and for a reddish sunset. Short wave lengths, that is the blue

spectrum, are more likely to be scattered perpendicular to the incoming direction of the light. The opposite holds for long wave lengths, that is the red spectrum. As a consequence, the zenith of the sky is more bluish than the horizon (compare Figure 5.1). For the same reason the sunlight traveling a long way through the atmosphere at sunset, appears to be orange. Since the blue spectrum is more likely to be scattered away from the viewing ray, the red component of the light is more likely to remain.

In Section 4.7 we have outlined the volumetric effects which are currently used in the actual game title AquaNox. Now we cover the volumetric algorithms which are known in general and judge them by suitability for real time rendering. We must keep in mind that all the algorithms presented in the following have been developed with the main directive to maximize rendering speed. Since realism is no strict aim in interactive entertainment it is often much easier to fake certain volumetric effects such as explosions and smoke. But with the increasing speed of the graphics accelerators and the CPU, there will be a growing demand for advanced rendering techniques. Right now some of the best-selling game titles like “Grand Theft Auto II”, whose successor is expected to yield earnings of more than 200 million dollars, are approaching the average budget of a major Hollywood film production and thus a certain realism of the renderings is expected. If this development is continuing, there will be great needs in the near future with respect to interactive realism. Right now, however, the collection of applied volumetric algorithms comprises mostly the following methods:

5.1 Sky Dome

The easiest approach to rendering clouds is the so-called sky dome. It is used if a cloudy sky has to be displayed with minimum rendering overhead. For that purpose a tessellated hemisphere is textured with a photograph of a real sky which is resampled in the polar coordinates of the hemisphere (see Figure 5.1). If the radius of movement of the viewer inside the dome is small compared to the size of the hemisphere then the perspective distortion of the sky dome remains small, so a sky dome is an effective way to fake a real sky. It is also easy to animate the clouds. The major drawback, however, is that one cannot move into the clouds, which for example is necessary in flight simulations.

5.2 OpenGL Fog

If the viewer is allowed to maneuver inside clouds or fog, then a sky dome is not adequate. Assuming constant gas density, the absorption of light on its way

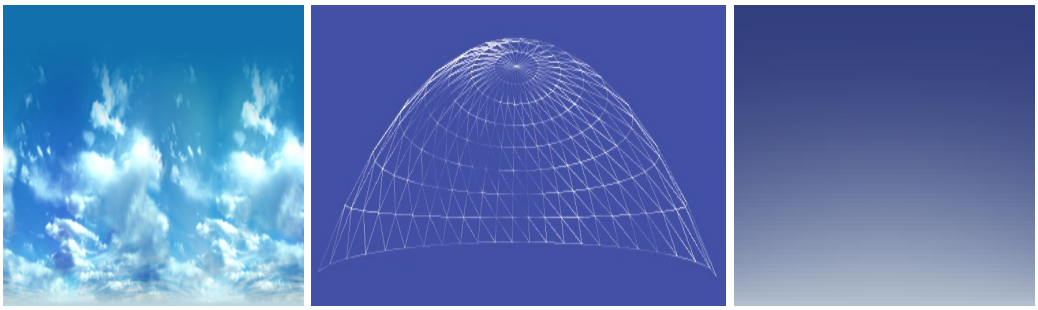


Figure 5.1: The triangle mesh of the sky dome hemisphere and two example sky dome textures resampled in polar coordinates. The zenith of the hemisphere corresponds to the upper edge and the horizon to the lower edge of the sky dome textures. The left sky dome is a stitched photograph while the right one is a synthetic texture. The latter mimics the effect that the sky usually is more bluish at the zenith due to wave length dependent scattering of the sun light (Rayleigh scattering).

through a medium leads to an exponential attenuation of the light intensity with respect to the traveled distance. Assuming constant in-scattering from the sun, the attenuated light intensity is increased again at each point on the viewing ray. Both effects together determine the appearance of fog. The longer the traveled distance through the medium the darker the background but the brighter the fraction of scattered light from the sun (see Figure 5.2).
This observation is the motivation for the so-called OpenGL fog [80], which is the simplest solution of the ray integral. For each rendered pixel the attenuation of the light intensity I of a surface patch is calculated by means of the following formula

$$I' = Ie^{-\tau z} \tag{5.1}$$

where z is the distance to the eye (the z-coordinate of each fragment) and τ is the optical density of the medium. For air the optical density is about 0.001 to 0.00001 depending on the actual weather conditions and other factors like humidity.

As mentioned above, absorption and emission occur together in nature. Absorption alone would render distant objects completely black. But since there is also atmospheric in-scattering, we need an additional term to express the tendency of the horizon to fade into white (see Figure 5.3 for an example of a simple foggy scene). In first approximation, the sun light scattered in from the atmosphere can be expressed as an ambient light, since it is mostly isotropic. This means that the absorbed light along a viewing ray is replaced by the light scattered uniformly

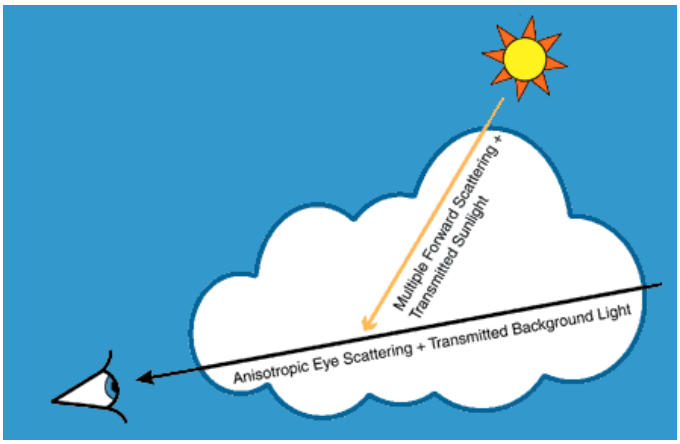


Figure 5.2: Absorption and scattering in an optical medium (from [37]).

from the sun into the direction of the ray. With this interpretation the RGB color components each fogged pixel are calculated as follows:

$$I'_{r/g/b} = 1 - (1 - I_{r/g/b})e^{-z\tau} \tag{5.2}$$

This formula can be evaluated efficiently by the graphics hardware. In fact, OpenGL-fog is implemented in all actual consumer graphics accelerators. OpenGL also supports other fog functions such as a linear attenuation. These additional functions are depicted on the right of Figure 5.3.

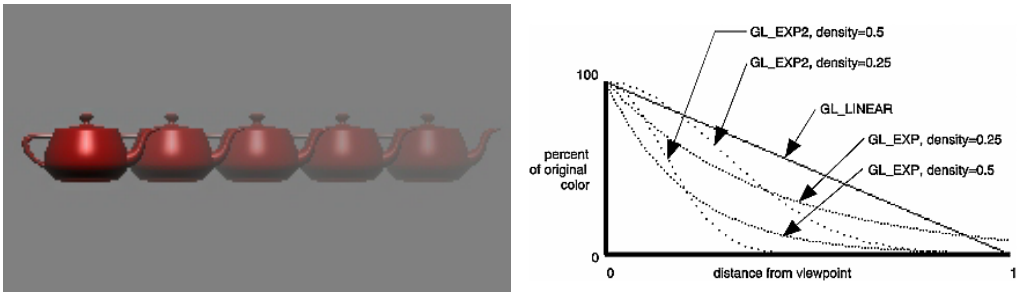


Figure 5.3: Plain OpenGL fog. **Left**: Fog illustrated by a depth ramp of the famous Utah teapot. **Right**: Fog functions as supported by OpenGL (from [80]).

5.3 Layered Fog

As seen above, the ray integral reduces to a mere exponential term for the case of a uniform gas density. For the general case, however, no analytic formulation is known and a numerical integration is required. But in some cases the integration or at least a large fraction of the work can be offloaded into a preprocessing step which is called pre-integration.

For the case that the gas distribution varies in a single dimension an explicit solution is known. If the density of the gas varies in the vertical dimension (see Figure 5.4), which means that it depends on elevation only, the ray integral depends on a total of three parameters: the height of the viewer, the height of each surface fragment and its distance in screen coordinates. Now the colors and opacities of the ray integral are defined by a three-dimensional function which can be pre-computed and stored in a 3D table, hence the table is said to be pre-integrated.

For each frame the slice of the 3D table that corresponds to the actual height of the viewer is put into a 2D texture, transfered to graphics memory, and used as a texture for all surface patches with the texture coordinates (s,t) of each vertex being set to its elevation and the distance in screen coordinates. So the problem is effectively reduced from 3D to 2D. The method is dubbed layered fog [57, 39] and is easily implemented as an additional 2D texturing pass in any game engine. The approach is per-pixel exact at the expense of not being able to define the gas distribution freely.

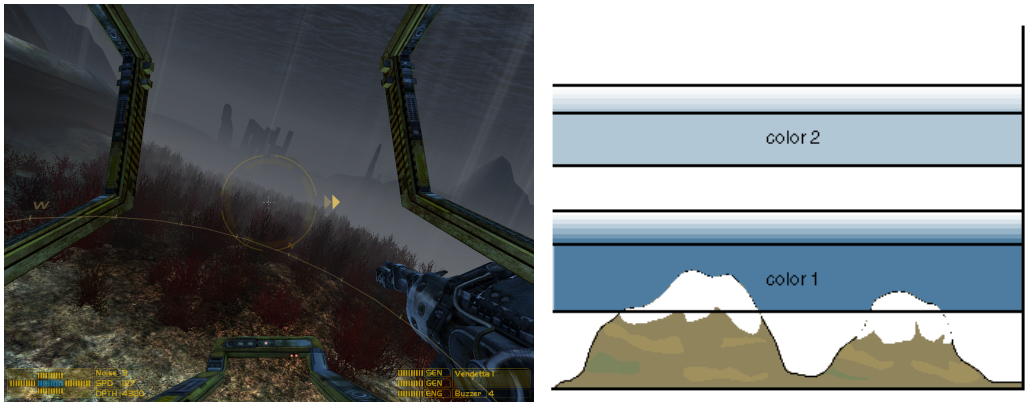


Figure 5.4: Layered fog as utilized in the sequel of AquaNox. The optical density of the fog depends on elevation only, such that mist is visible only in the lowlands of the landscape (left) or as multiple cloud layers (right).

5.4 Bounded Layered Fog

As opposed to the use of OpenGL and layered fog which require only one inexpensive operation per pixel, more sophisticated volume rendering methods usually lead to multi-pass rendering algorithms. One of those multi-pass algorithms is bounded layered fog as proposed by Mech [70]. Bounded layered fog, which sometimes is also called patchy fog, is defined as a volume with a sharp boundary and a constant gas density inside the boundary. The outer hull of the volume is defined by a triangle mesh. This mesh is rendered twice per frame in the following fashion: First the back facing triangles are rendered into the frame buffer with additive blending enabled and the color of each vertex set to its distance to the viewing plane. Then the same procedure is repeated for the front facing triangles with subtractive blending enabled. This effectively computes the intersection lengths of each viewing ray with the volume.

In the third and last pass the intersection lengths stored in the frame buffer are transformed into attenuation factors using pixel textures [112]. Since the intersection length can grow arbitrarily large a high resolution frame buffer is mandatory to prevent Mach bands. In Section 10 we present an extension to Mech's method which does not show this restriction and is more flexible with respect to defining the fog boundary.

With the upcoming of programmable PC graphics hardware the accuracy of the method can be increased by using a floating point render target to compute the intersection lengths. Then ping-pong filtering [74] can be applied to map the intersection lengths to the attenuation factors. For this purpose, a 1D dependent texture is used which contains the exponential attenuation function according to Equation 5.1.

Even though Mech's method is a multi-pass algorithm it is reasonable fast if the hull of the fog volume is not too complex. Otherwise the majority of the pixels tend to be rendered multiple times, thus the overdraw is considerable. Only sharp boundaries and a constant fog density are possible. This does not correlate with reality where the density of a cloud usually varies. In addition, the method does only feature approximate lighting, hence the appearance of bounded layered fog is quite artificial (compare Figure 5.5). Nevertheless its performance makes it a good choice for interactive entertainment.

5.5 Billboards

As mentioned in Chapter 4 a common method to render "fuzzy" volumetric effects is to use billboards. Billboards are just quadrilaterals which are aligned to face towards the viewer. They are textured with custom made images of mostly trees,

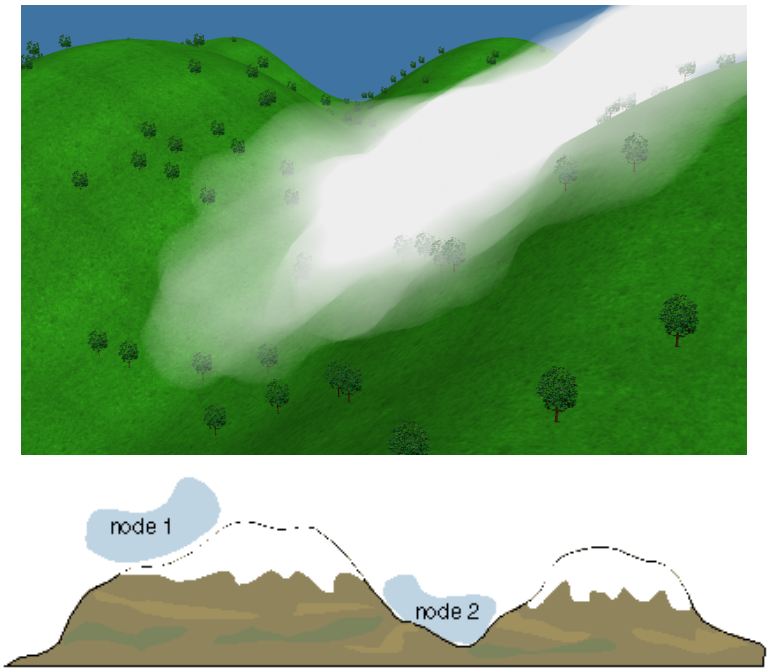


Figure 5.5: Example and schematic view of bounded layered fog.

explosions, or smoke. Explosions, for example, are usually made up of several evolving blast elements that are blended over each other. Despite the simplicity of this approach the effects can be quite stunning. If the billboard textures are designed by an experienced graphics artist explosions can look very impressive. Unless the view point is very close to the center of the explosion the billboards is not unveiled. The billboard method is fast and simple but it is not suited very well for the display of natural gaseous phenomena as we will see in the following.

5.6 Metaball Methods

A sibling of the billboard technique is the metaball method for the display of clouds. Instead of billboards it uses spheres which are textured with a projection of the volume each metaball is associated with. The first approach to incorporate this technique was presented by Gardner [31] as early as in 1985. In fact, this method was the first that achieved interactive frame rates while maintaining reasonable quality. On the one hand, the metaballs can be cached efficiently using vertex arrays, but in comparison to billboards more triangles are needed per graphics primitive. On the other hand, more detailed clouds can be constructed easily by

clustering several metaballs (see Figure 5.6). More recently, Elinas et al. [25] managed to render highly detailed natural clouds, but as they note in their paper the metaball technique is not suitable for a fly-through, since the metaballs clearly become observable in the vicinity of a cloud.

Figure 5.6: A cumulus cloud defined by a cluster of metaballs (from [31]).

5.7 Impostor Based Methods

In order to compensate for the disadvantages of the metaball method impostors are utilized frequently. They were first introduced by Schaufler et al. [95, 97] to speed up the display of objects with complex geometry. Impostors can be thought to be dynamic billboards. An impostor is essentially a billboard with an associated polygonal object which is used to generate the billboard texture for a specific point of view. So the texture of an impostor adapts to the actual point of view. If the deviation of the cached texture from the projection of the underlying geometry exceeds a specific error threshold, the impostor texture is recomputed from the original polygonal data. The deviation is calculated by taking the maximum of the projective shifts of each polygon on the impostor plane induced by the movement of the viewer. Speaking non-technically, the deviation of the impostor texture is the projected distance between where each polygon is placed and where the viewer observes it on the impostor texture. The probability of an update of the impostor texture is proportional to the movement of the viewer since the last capture. The update probability is also inversely proportional to the viewing distance.

The impostor technique converts complex polygonal objects into an image based representation. The speed up is due to the fact that the billboard can be rendered much faster than the object itself which may consist of thousands or even millions of polygons. However, the impostor texture has to be recomputed whenever the perspective distortion grows too large. In this case the polygonal object has to be rendered and transformed into a texture. So no speed up is achieved for the frame in which the impostor is updated. In subsequent frames the cached impostor texture can be reused to obtain a significant speed up until the impostor has to be updated again. Hence, updates need not occur too frequently to keep a high performance level. More precisely, the impostor method achieves a significant speed up compared to polygonal rendering if the following criteria are met:

- The triangle count of the cached polygonal geometry is large compared to the geometry of the billboard frame which consists of two triangles (one quadrilateral).
- The cached textures completely fit into texture memory.
- The mean update frequency of the impostors is significantly less than once per frame.
- The visible impostors in the scene need not be updated simultaneously.

The impostor method has been successfully applied to speeding up the rendering of large polygonal models and afterwards to speeding up cloud rendering. Here, each impostor represents a small cloud. Its appearance is computed by light scattering methods (see Chapter 6 for more details). For a crowd of cumulus clouds, for example, the scene can be represented with only a few impostors (see Figure 5.7). In this case the performance is quite impressive. However, the update rate of the impostors increases with the proximity to a cloud. Nearby or inside a cloud the impostor texture has to be recomputed almost every frame, so that the speedup is nearly zero. Furthermore, the update of an impostor manifests itself in a temporal aliasing artifact. These drawbacks can be avoided only by switching to a real volumetric representation of the clouds.

Nevertheless, the impostor method is used widely for cloud rendering due to the realistic appearance of the clouds. The most notable papers here are those of Dobashi et al. [16] and of Harris et al. [37]. Dobashi achieves very high image quality including the depiction of soft shadows and shafts of light, but rendering times are in the range of several seconds (the rendering time of the image in Figure 5.8 was about 20 seconds).

Harris' method performs significantly better with true real time performance if the number of impostors is kept reasonable small. Neither soft shadows nor shafts

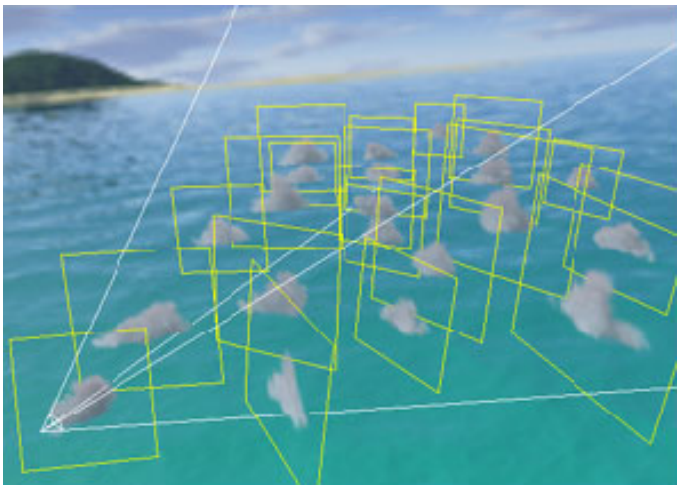


Figure 5.7: A crowd of cumulus clouds represented by impostors (image from [37]).

of light are included (see Figure 5.9). More detailed scenes like entire cloud layers require a larger number of impostors, hence result in lower frame rates.

In summary, the impostor method can be utilized not only to speed up the display of large polygonal models but also to accelerate cloud rendering. Instead of diving into the algorithmic details of both Dobashi's and Harris' method, the methods are very well suited for their specific application area (see also Chapter 12), but have one major drawback which is discussed in the following.

As the main drawback, the impostor method works well in the case of what we call good weather conditions, that is a crowd of small cumulus clouds. But if large cirrus clouds, overcast sky, or huge layers of mist have to be dealt with impostors there arise a variety of problems. Large clouds require large impostors which exhibit large projective distortion. Therefore the update rate is high, so that the speedup is dissatisfying. But if we try to make up larger clouds from several smaller impostors, rendering artifacts occur whenever the impostors overlap each other. In summary, large cloud formations are difficult to deal with impostors. Impostor methods work well for good weather conditions but they are not suited for the visualization of arbitrarily shaped clouds and in particular for the visualization of weather simulation data. Here real volumetric algorithms are needed. In order to come up with a solution for the mentioned problems we take an excursion into volume visualization in the next chapter to see what can be learnt from this research area.

Figure 5.8: Dobashi's cloud rendering method (image from [16]).

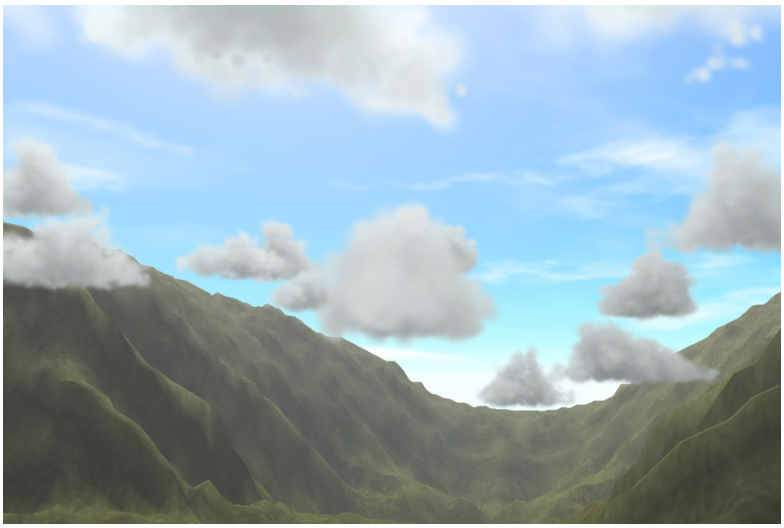


Figure 5.9: Harris' cloud rendering method (image from [37]). Note that the distant cirrus clouds are rendered using a sky dome.

Chapter 6
Volume Rendering: The Basics

In this chapter we describe the foundations of volume rendering. Our special interest lies on revealing possible improvements for the real time display of clouds.

6.1 Basic Principles

The principle of volume rendering is based on the physical model of an optical medium, which typically is a gas or a liquid. The density of the medium is defined as a three-dimensional scalar function. Light traveling through the so-defined volume is scattered and absorbed as single photons hit the atoms of the medium. The probability of hitting an atom is proportional to the gas density. In case of a hit the energy of the photon is either absorbed and transferred into thermal energy or the photon is sent out again in a more or less random direction. In the latter case the wave length of the photon may also shift. The appearance of natural gaseous phenomena is thus the result of a vast number of photons being scattered back and forth multiple times until they finally reach the observer. Due to the complex paths of scattered photons the exact solution of the observed light properties is rather time consuming.

Hence it is no surprise that for traditional rendering, the presence of an optical medium is usually neglected. Thus, scattering is assumed to appear only at the surface of an object. Even though this is a strong simplification, the correct simulation of the physical phenomena, that is of light scattering off various types of surfaces, remains a challenging task. As there is a large number of solutions to the traditional rendering challenge we only introduce the basic concept of the rendering equation in this chapter.

6.2 The Rendering Equation

The rendering equation of Kajiya [44] subsumes a wide variety of rendering algorithms and provides a unified context for viewing them as more or less accurate approximations to the solution of a single equation.

It has to be mentioned that the idea behind the rendering equation originates from the area of material sciences. A description of the phenomenon simulated by this equation has been well studied in the radiative heat transfer literature for years [102].

The rendering equation is

$$I(x,\omega) = I_e(x,\omega) + \int_{\Omega_+} I(y(x,\omega'),-\omega')\, f_r(\omega',x,\omega)\cos\theta' d\omega'. \qquad (6.1)$$

where:

$I(x,\omega)$	is related to the intensity of light passing from point x into direction ω
$y(x,\omega')$	denotes the point y that is visible from x in direction ω'
$I_e(x,\omega)$	is related to the intensity of emitted light from point x in direction ω
$f_r(\omega',x,\omega)$	is related to the fraction of light scattered from ω' into direction ω by a patch of surface at x
θ	is the angle between the surface normal at point x and the direction ω'

The equation is very much in the spirit of the radiosity equation, simply balancing the energy flows from one point of a surface to another. The equations state that the transport intensity of light from one surface point to another is simply the sum of the emitted light and the total light intensity which is scattered toward x from all other surface points. The equation differs from the radiosity equation because, unlike the latter, no assumptions are made about reflectance characteristics of the surfaces involved. As opposed to this, the radiosity equation presumes a solely diffuse reflectance behavior.

As an approximation to Maxwell's equation for electromagnetics the rendering equation does not attempt to model all interesting optical phenomena. It is essentially a geometrical optics approximation. It only models time averaged transport intensity, thus no account is taken of phase in this equation – ruling out any treatment of diffraction. In addition, no wavelength or polarization dependence is mentioned explicitly in the rendering equation. Finally, it is assumed that the media between surfaces is of homogeneous refractive index and does not itself participate in the scattering light. Treatments of participatory media and of phase and diffraction can be handled with path integral techniques. For instance, an integro-differential equation is necessary for participating propagation media [45].

6.3 The Ray Integral

As mentioned in the last section the rendering equation of Kajiya does not include the effects of participating media. In the following we account for the change of the light transmitted through the participating medium from point y to x. The formula, which describes the absorption, the scattering, and the emission of light on its the way through the medium is called the ray integral.

Let $I(x,\omega)$ be the intensity at position x in direction ω, and let $k_t(x)$ be the extinction coefficient of the participating medium. This is the total opacity (absorption plus scattering) per unit length so $k_t(x)I(x,\omega)ds$ is the intensity removed along an infinitesimal ray segment ds at x. Let the albedo a be the fraction of the removed intensity scattered into other directions, and let the phase function $f_p(\omega,\omega')$ be the directional distribution function for this scattered intensity, so that $\int_B f_p(\omega,\omega')d\omega$ is the fraction of the scattered intensity from direction ω' that ends up in solid angle B. Then

$$ak_t(x)ds\int_{4\pi} I(x,\omega')\,f_p(\omega',\omega)d\omega' \tag{6.2}$$

is the intensity scattered into the direction ω along the ray segment ds from other directions ω' in the 4π unit sphere. This is the source function (compare [102]) in the absence of volume emission. The integro-differential equation for $I(x,\omega)$ including emission is thus

$$\frac{dI(x,\omega)}{ds} = -k_t(x)I(x,\omega) + ak_t(x)\int_{4\pi} I(x,\omega')\,f_p(\omega,\omega')d\omega' + I_e(x,\omega) \tag{6.3}$$

Using an integrating factor (see [102, 119, 92], this can be integrated along a path $x'(s) = x + s\omega$ from $x = x'(0)$ to $y = x'(s_0)$ at the edge of the medium, to give the ray integral:

$$\begin{aligned} I(x,\omega) \;=\; & I(y,\omega)exp\left(-\int_0^{s_0} k_t(x'(s))ds\right) + \\ & \int_0^{s_0}\left(I_e(x,\omega)exp\left(-\int_0^{s} k_t(x'(t))dt\right)\right)ds + \\ & a\int_0^{s_0}\left(k_t(x'(s))exp\left(-\int_0^{s} k_t(x'(t))dt\right)\int_{4\pi} I(x'(s),\omega')\,f_p(\omega',\omega)d\omega'\right)ds \end{aligned} \tag{6.4}$$

Substituting the ray integral for $I(y(x,\omega'),-\omega')$ in the rendering equation yields the complete equation that both takes scattering at surfaces and inside a participating medium into account. For volume rendering alone the interaction with surfaces is not necessary, so a solution of the ray integral suffices.

6.4 Light Scattering in Participating Media

Now that the fundamental properties of light transport and scattering have been laid out the question surely is how to render images efficiently? To recall the complexity of the entire problem, for each ray the integrated intensity consists of the energy scattered in the direction of the ray. All these scattered energies along the ray again consist of scattered energy originating from other paths. All these energies must be collected properly to yield a natural looking cloud, for example. In particular, it is not sufficient to collect only the light scattered directly from the sun into the direction of the viewing ray. This precondition is illustrated in Figure 6.1 where both single and multiple scattering are compared against each other. If the energy collection process is performed only a single time (hence single scattering) the clouds clearly look unacceptable in comparison to multiple scattering.

The phase function $f_p(\omega,\omega')$ for scattering in air is not isotropic, that is the energy scattered back is smaller than the energy scattered forward. Figure 6.2 shows the difference between anisotropic and isotropic multiple scattering. Using an isotropic approximation of the real scattering conditions in nature allows to precompute the light intensities, since isotropic scattering is view-independent. In contrast, anisotropic scattering is view-dependent, so the light scattered into the viewing direction has to be recalculated for every frame.

In our opinion isotropic scattering is sufficient in many cases. The difference mainly becomes observable when the sun is visible directly behind a cloud or beneath the edge of a cumulus cloud. For thick cloud layers the distinction between isotropic and anisotropic scattering is not so important. For each application scenario it has to be decided whether or not anisotropic scattering is worth the additional effort.

Nevertheless, for atmospheric rendering and skylight exact solutions of the ray integral, hence anisotropic scattering models are necessary. Due to wavelength dependent Rayleigh scattering the sky color is shifted either to red or blue tones [17].

Figure 6.1: **Top**: Single scattering, **Bottom**: Multiple scattering (images from [37]).

6.5 Rendering Solutions for Participating Media

Holly Rushmeier [93, 92] applied two techniques to solve the general problem of volume rendering with participating (i.e. absorbing, emitting, and scattering) media. One is the Monte Carlo method, where a random collection of photons or flux packets are traced through the volume, undergoing random scattering and absorption. This method can accurately model all the physics of scattering, but may take an impractical number of random trials to converge to a useful solution.

The other one is the zonal method [93] for isotropic scattering, which divides the volume into a number of finite elements which are assumed to have constant radiosity. This requires the calculation of a form factor between every pair of elements. With the assumption of the Galerkin finite element scheme and an interactive method for solving the resulting matrix equation the total computational cost is $O(n^7)$ for a cube of n^3 elements.

Although the two mentioned methods accurately compute the solutions of the general ray integral the restriction to isotropic scattering and in particular the slow runtime behavior disqualifies the methods for real time applications.

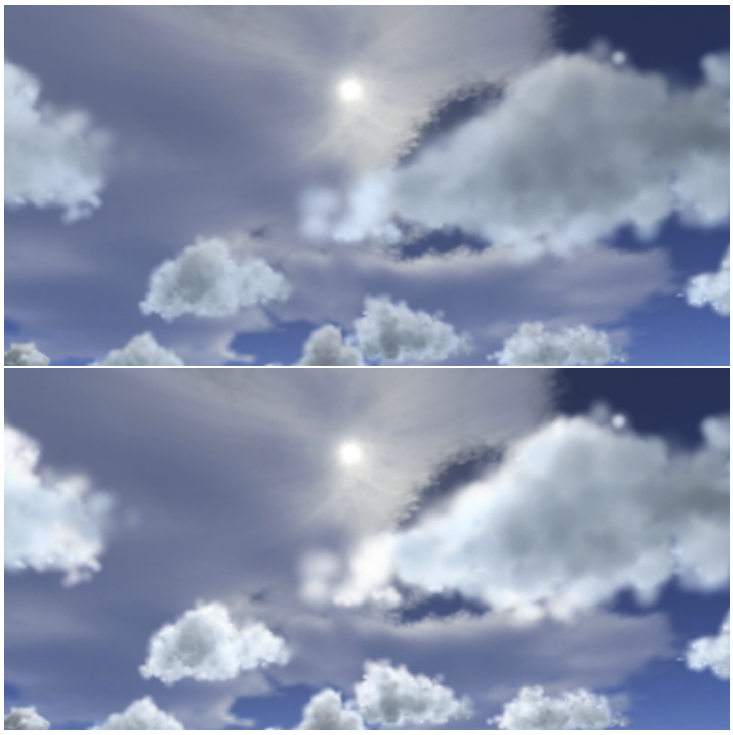


Figure 6.2: **Top**: Isotropic scattering, **Bottom**: Anisotropic scattering (images from [37]).

In contrast to the previous methods, the discrete ordinates method in radiation transfer (see [102]) achieves a large speed up with regular cubical grids. Since the method makes essential use of the homogeneity of the grid, it will not work on more general finite element meshes. One also has to point out that this method produces the so called ray effects (a type of aliasing artifacts), because it is equivalent to shooting the energy from an element in narrow beams along the discrete directions, missing the regions between them. Max [67] presented an approximation to the discrete ordinates method, which reduces the ray effect by shooting radiosity into the whole solid angle (see also [43]). This approach has been improved continuously (compare [78]), however the basic idea remained the same over time.

As a concluding remark we emphasize that due to the inherent complexity of the scattering process a visual simulation of clouds is restricted to application scenarios which only require a local solution of the ray integral. At present the real time simulation of a global system seems impractical. For the simulation of such a global system we need to further simplify the ray integral equations, which leads to a volume visualization technique known as direct volume rendering.

Chapter 7
Direct Volume Rendering

For volume visualization purposes scattering is usually neglected, since it is computationally expensive. In other words, the albedo a in Equation 6.4 is assumed to be zero. Additionally it is assumed that only a volume describing the density of the optical medium is present in a scene, so that there is no back scattering of surfaces. Then Equation 6.4 is reduced to the integration of a single light ray:

$$I(x,\omega) = \int_0^\infty \left(I_e(x,\omega) exp\left(-\int_0^s k_t(x'(t))dt \right) \right) ds \qquad (7.1)$$

The evaluation of the above line integral is characteristic for all direct volume rendering techniques [18]. In comparison to indirect volume rendering methods like isosurface extraction no intermediate representations are generated.

One of the classic direct volume rendering scenarios is the medical visualization of computer tomography data. Here the interactive and intuitive examination of the scanned organs and not the photorealistic appearance thereof is of prime importance. This makes direct volume rendering the method of choice for medical visualization.

On the other hand, it has to be mentioned that, of course, the true appearance of clouds cannot be reproduced with direct volume rendering techniques. But in Section 11 we will see that the seemingly restrictive physical model, in fact, can be extended to mimic the natural appearance of clouds. Thus the advantages of fast rendering and realistic display can be combined.

7.1 Transfer Functions

The density of the optical medium is usually described by a three-dimensional scalar function $f(x,y,z)$. In order to assign a specific appearance to the scalar values a so-called transfer function is used. The emission and the optical density of the medium are given by the transfer functions denoted by $\kappa(f(x,y,z))$ and $\rho(f(x,y,z))$, respectively.

Regarding the physical nature of light transport, both absorption and emission are related to the density of the medium, that is the more dense the medium the more light is being emitted and absorbed. If the medium is fully transparent

neither absorption nor emission are being perceived. In order to reflect this physical relationship, the emission κ is often implicitly pre-multiplied with the density function ρ. In this case one speaks of a pre-multiplied transfer function.

7.2 Grid Types

Since in the majority of cases the volume cannot be represented as a continuous function, the domain of the scalar function is usually discretized. We categorize these discretized representations into three main classes: Regular, curvilinear and unstructured grids. In the first case the domain is a three-dimensional matrix which is sampled uniformly in all three principal axis. If the sampling is not uniform one speaks of a rectilinear grid. Semi-regular grids like octrees or multigrid meshes are special cases of a regular grid. In the second case the domain is still a regular matrix in computational space, but in object space it may have any shape. To give a example, a regularly meshed cylinder is a curvilinear grid. The remaining grid types are covered with the last case of unstructured grids. In contrast to surface meshes which are constructed from convex polygons (triangles, quadrilaterals, etc.) unstructured grids are built from convex polyhedra, such as hexahedra, prisms, pyramids, or as the most common case, tetrahedra. To name only a few typical application areas regular grids are generated by CT or MR scanners, while unstructured grids are mostly encountered as the mesh type of numerical finite element simulations.

7.3 Ray Casting

The neglection of scattering has the advantage of a fast solution of the ray integral as described in the following: By shooting a viewing ray through every pixel in image space into the volume the light intensity of each pixel is simply equivalent to the incoming light collected on the corresponding light ray. So the task of direct volume rendering is to calculate the line integral as given above for each pixel in image space. In general, even this seemingly simple line integral cannot be solved analytically. Instead, a numerical integration is required, that is the line integral is approximated by a Riemann sum. This means that the volume is sampled at equally spaced points on the light ray. This basic procedure is called ray casting (see Figure 7.1 for a schematic view).

The emissions associated with each sample point are attenuated on the way to the viewer due to the absorption of the medium. From one sample point to another the attenuation can be expressed in terms of the mean optical density τ between the two respective sample points. Let d be the distance between the sample points x_i

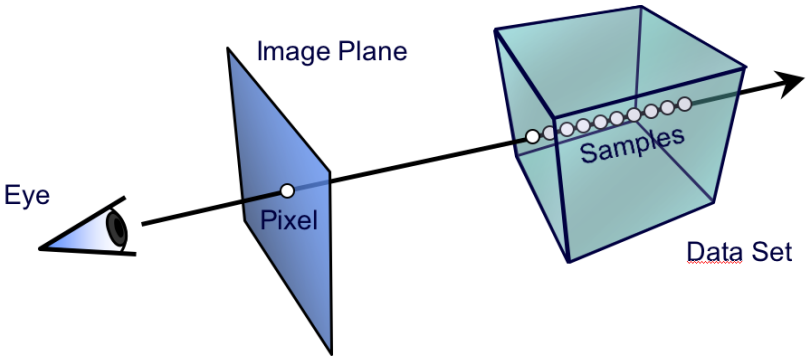


Figure 7.1: Basic principle of ray casting.

and x_{i+1}, then the mean optical density is given by $\tau_i = \frac{1}{2}(\rho(f(x_i)) + \rho(f(x_{i+1})))$. Hence, the (approximated) opacity α_i of the corresponding ray segment is $1 - exp(-\tau_i d)$. Similarly the average emission of the ray segment is given by $E_i = d\frac{1}{2}(\kappa(f(x_i)) + \kappa(f(x_{i+1})))$. By processing the samples on the light ray in a back to front fashion the final color C of each pixel is reconstructed by summing up the emissions using the following blending formula:

$$C_{i+1} = (1 - \alpha_i)C_i + E_i$$

For a decreasing ray segment length d this approximation converges to the exact solution of the continuous formulation of the line integral. In practice, the scalar density function f is band limited in almost any case. Thus, the sampling distance d can be chosen so that it corresponds to half the wave length of the highest frequency in the frequency domain representation of the scalar function (often referred to as the Nyquist frequency). For instance, the minimum required sampling distance for a regular grid is the edge length of the voxels, so that each contributing cell is touched at least once on the viewing rays. For unstructured grids with highly varying cell sizes the sampling distance is usually not set to a constant value, but rather adapted to the length of the intersections of the viewing rays with each cell.

Recently, hardware-accelerated implementations have been presented both for structured [88] and unstructured [110] volume data exploiting programmable graphics hardware. The achieved speed-ups are quite remarkable in comparison to a pure software approach, but there are still some limitations such as texture memory size, which currently restrict the application of hardware-accelerated ray casting. But with the upcoming of future improved consumer graphics hardware those approaches clearly will become a very interesting alternative to the traditional methods which are described in the next sections.

7.4 Slicing via 3D Textures

For regular grids there exist a variety of acceleration techniques which improve the performance of the basic ray casting approach. These are mainly early-ray termination, space leaping, the shear-warp algorithm [54], 3D slicing or texturing [1, 7, 113, 26], and splatting [115, 14]. Early ray termination means that rays shot into the volume may stop in case of reaching full opacity, since the remaining ray is occluded entirely. Likewise, entirely transparent areas can be skipped over very quickly, which is called space leaping. The shear-warp is a pure software approach, which takes advantage of the fact that an orthogonal projection of a regular volume can be decomposed into two consecutive shear operations in image space, so that resampling and interpolation of the grid can be performed efficiently by 2D operations. As opposed to this pure software approach the hardware-accelerated techniques try to offload the computationally expensive resampling of the volume onto the graphics hardware. For this purpose a regular volume is packed into a 3D texture. Then view-plane aligned slices are drawn from back to front as illustrated in Figure 7.2. For each slice the graphics hardware is setup to interpolate the correct colors and opacities from the 3D texture. In this way the entire set of viewing rays is treated simultaneously, resulting in a tremendous speedup due to the high rasterization performance of the graphics accelerator.

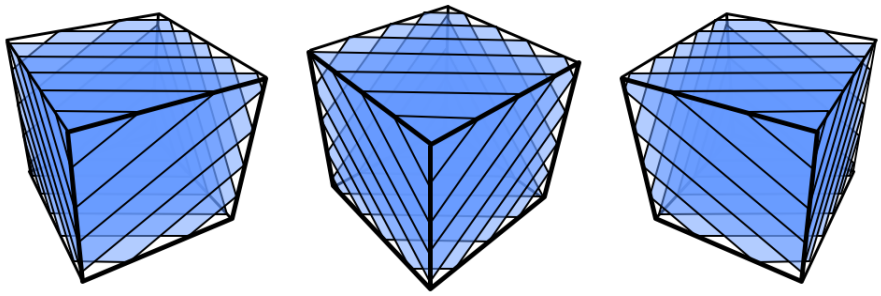


Figure 7.2: Hardware-accelerated volume rendering via 3D textures.

While this method efficiently utilize the graphics hardware, it usually has problems coping with large datasets, since the 3D texture has to reside in dedicated texture memory. Data sets that do not fit entirely into texture memory need to be “bricked”, that is they are broken down into several smaller blocks that are rendered subsequently. However this method suffers from a slow transfer speed between the main and the graphics memory, because each brick has to be uploaded in each single frame. Hierarchical methods [55, 111, 4, 36] for volume compression are also known as a solution for insufficient texture memory, how-

ever these techniques will not be discussed in detail in this thesis, since these approaches have problems with maintaining a conforming view-dependent mesh. Recently, advanced lighting and pre-integration techniques [71, 48, 49] have been introduced which improve image quality but do not solve the problem of restricted texture memory size, so these approaches are also not discussed in this thesis.

As the last basic acceleration technique splatting should be mentioned here. In contrast to the previous methods splatting is not an image but an object space technique which approximates the footprint of each voxel with a splat kernel. The splatting algorithm processes all voxels in visibility sorted order and accumulates the splats on the image plane (or sheet buffer). The main drawback of the splatting algorithm is that the splat kernel is only a more or less coarse approximation of the true footprint of a voxel. As a result, the generated images look smoother in comparison to the results of a ray caster.

In general, the achieved frame rates of classic volume rendering techniques do not meet the real time demands of cloud visualization. To achieve higher frame rates we aim to exploit the inherent view-dependent nature of the cloud rendering problem. Our final goal is to develop a hierarchical view-dependent rendering algorithm that displays the clouds nearby with high detail and distant clouds with less detail. For this purpose we first have to dive into the topic of unstructured volume rendering.

Although a variety of hardware-accelerated methods are known for regular volumes, unstructured grids did not yet profit as much from the upcoming of programmable graphics hardware. In the two chapters we develop a technique based on hardware-accelerated cell-projection that closes this apparent gap.

Chapter 8
Pre-Integrated Cell-Projection

More than ten years ago direct volume rendering of unstructured tetrahedral meshes was dramatically accelerated by the Projected Tetrahedra (PT) algorithm by Shirley and Tuchman [101], which is summarized in Section 8.2. Although there are numerous competing approaches to direct volume rendering of unstructured meshes, e.g. ray casting [103, 110], slicing [127], or sweep-plane algorithms [112], several aspects of the PT algorithm are still subject of current research, e.g. the visibility sorting of tetrahedral cells (see [13, 28] and references therein). Our extensions of the PT algorithm are restricted to the rendering of projected tetrahedra.

8.1 Visibility Sorting

For this work it is assumed that the visibility ordering of an unstructured mesh has already been computed. A visibility (or depth) ordering of a set of polyhedral cells is an ordering such that a cell A precedes a cell B if B occludes A. This results in a back to front ordering of the cells. For semi-regular grids like octrees the visibility sorting is trivial (see Section 11.5), but for unstructured meshes the depth ordering has to be computed explicitely. This can be achieved in $O(n)$ time for convex simplicial grids by using the MPVO algorithm of Williams et al. [118]. It starts with an arbitrary cell and sorts its adjacent cells by inserting them either at the beginning or the end of a queue depending on whether the neighbors are attached to either a front or a back face. By recursively repeating this procedure for all inserted cells a depth ordering is constructed. Non-convex or disconnected grids are sorted only partially by this method. In order to obtain a correct ordering, occlusion relations between the boundary faces have to be established. The so-called BSP-XMPVO algorithm [13] computes these relations by searching them in a BSP (binary space partition) tree. It is commonly assumed that the cells do not overlap in a cyclic way, otherwise the dependency graph cannot be sorted in a unambiguous way. Fortunately most data sets encountered in practice do not have cycles, but in the case one encounters a cycle the algorithm of Kraus et al. [51] renders a cyclic group consisting of n tetrahedra in $O(n^2)$ time.

8.2 The Original PT Algorithm

The PT algorithm visualizes a scalar function $f(x,y,z)$ defined over a region of three-dimensional space by rendering partially transparent polygons, which can be processed very quickly by specialized graphics hardware. The PT algorithm can be summarized as follows (see also [101]):

1. Decompose the volume into tetrahedral cells. For example, a prism is decomposed into three tetrahedra. Scalar values are defined at each vertex of the mesh. Inside each tetrahedral cell, $f(x,y,z)$ is assumed to be a linear combination of the vertex values.

2. Sort the cells according to their visibility.

3. Classify each tetrahedron according to its projected profile and decompose it into smaller triangles (see Figure 8.1).

4. Find color and opacity values for the triangle vertices using ray integration.

5. Render the triangles.

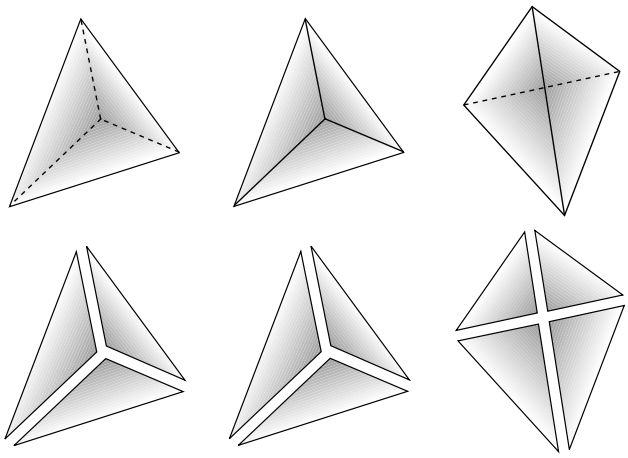


Figure 8.1: Classification of non-degenerate projected tetrahedra (top row) and the corresponding decompositions (bottom row) according to [101].

8.3 Drawbacks of the Original PT Algorithm

The original PT algorithm approximates the opacity and color between vertices linearly resulting in Mach bands as reported by Max et al. in [66]. Stein et al. presented a solution for the correct interpolation of opacities utilizing 2D texture mapping in [106], which is also discussed in Section 8.2. However, this method is restricted to linear transfer functions for the opacity and still interpolates color components linearly ignoring the transfer functions for them inside the tetrahedra.

Our first improvement of the PT algorithm allows us to render both, opacity and color, accurately by exploiting 3D texture mapping. In particular this method allows us to employ arbitrary transfer functions. The method and its application to a volume density optical model is described in Section 8.4. In Section 8.5 we derive an approximate rendering method based on 2D texture mapping, which is supported by considerably more graphics systems and requires less texture memory. A second extension allows us to include the rendering of isosurfaces in the PT algorithm using 2D texture mapping without extracting a polygonal representation of the isosurfaces. There are numerous algorithms to display isosurfaces efficiently. We will mention a selection in Section 8.6. However, none of these algorithms takes any particular advantage of the PT algorithm. Therefore, the costs of displaying an isosurface were not reduced by a combination with the PT algorithm in the past.

Our approach, however, reuses the visibility ordering and the decomposition of the tetrahedral cells, which are an essential part of every variant of the PT algorithm. The visibility ordering algorithms described in Section 8.1 all appear to be compatible with our rendering extensions. By reusing the ordering and decomposition of tetrahedra our method is capable of rendering isosurfaces without constructing a polygonal representation. As it is conceptually similar to the first pass of the multi-pass algorithm for smoothly shaded isosurfaces by Westermann and Ertl [113], we present a variant of this first pass in Section 8.7. We employ this idea in the context of the PT algorithm and present a specialized single-pass algorithm for flat-shaded isosurfaces using 2D texture mapping in Section 8.8. Moreover, a two-pass algorithm for smoothly shaded isosurfaces is described in Section 8.9.

Extensions for colored and multiple isosurfaces are discussed in Section 8.10, while Section 8.11 presents two methods for mixing isosurfaces with projected volume cells, either approximately but smoothly using appropriate blending and texture mapping or more accurately by modifying the texture maps.

We emphasize that the worst-case time complexities of all our methods, i.e. volume rendering with arbitrary transfer functions, rendering of multiple and smoothly shaded isosurfaces, and mixing of isosurfaces with projected volume cells, are linear in the number of tetrahedra and neither depend on the transfer

functions nor on the number of isosurfaces. In the reminder of this section and in Sections 8.4 and 8.5 we will only discuss methods to improve the last two points: ray integration and rendering of the decomposed triangles with emphasis on hardware-accelerated rendering.

8.4 PT with Accurate Opacity and Color

The original PT algorithm interpolates color and opacity linearly between the triangle vertices. This, however, is an approximation which leads to rendering artifacts as demonstrated in [66, 106].

In order to avoid these artifacts Stein et al. suggested in [106] to use a 2D texture map with the texture coordinates being the averaged extinction coefficient τ and the thickness l of the projected cell, while the texture map contains an α-component which is set to $\alpha = 1 - \exp(-\tau l)$. In between the vertices of each triangle the texture coordinates and, therefore, τ and l are interpolated linearly; thus, this approach is restricted to a linearly varying extinction coefficient τ, i.e. a linear transfer function $\tau = \tau(f(x,y,z))$. Moreover, the color is still linearly interpolated between vertices. Williams et al. extended these ideas to piecewise linear transfer functions in [120].

In this section a generalization of the method of Stein is presented which works for color and opacity, and places no restrictions on the transfer functions. We achieve these benefits by employing 3D texture mapping.

Let us start by investigating the situation depicted in Figure 8.2.

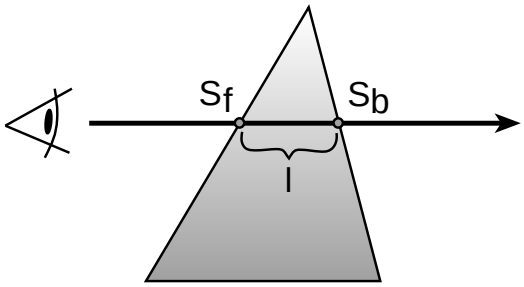


Figure 8.2: Intersecting a tetrahedral cell with a viewing ray. s_f and s_b are the scalar values on the entry (front) and exit (back) face respectively; l denotes the thickness of the cell for this ray.

As texture coordinates are interpolated linearly, we should only use variables, the values of which vary linearly with screen coordinates. We will restrict our con-

siderations to orthographic projections. In this case l varies linearly for geometric reasons; s_f and s_b vary linearly because they are interpolated linearly between vertices as mentioned above. Therefore, s_f, s_b, and l should be the three texture coordinates. Fortunately, all other values, e.g. color, opacity, etc., can be calculated from l, s_f, and s_b. Thus, we can set up a 3D texture map which contains the color and opacity characterizing the intersection of a ray and a cell in dependency of l, s_f, and s_b.

For many applications the calculation of the texture map is a preprocessing step and, therefore, not time-critical. Usually it includes a numerical integration of a ray for each texel in the 3D texture map as outlined in Section 7.3. We sketch the procedure for the volume density optical model proposed by Williams and Max [68, 119, 120] with a chromaticity vector $\kappa = \kappa(f(x,y,z))$ and a scalar optical density $\rho = \rho(f(x,y,z))$, which are the transfer functions of this model.

Assuming cells are processed back to front, the addition of the projection of a cell changes an existing pixel color I to a new pixel color I' by the formula

$$\begin{aligned} I' &= \underbrace{\int_0^l \exp\left(-\int_0^t \rho(s_l(u))\mathrm{d}u\right)\kappa(s_l(t))\rho(s_l(t))\mathrm{d}t}_{\mathrm{RGB_{t3D}}} \\ &\quad + \underbrace{\exp\left(-\int_0^l \rho(s_l(t))\mathrm{d}t\right)}_{1-\alpha_{\mathrm{t3D}}} \times I \end{aligned} \tag{8.1}$$

with the abbreviation

$$s_l(x) = s_f + \frac{x}{l}(s_b - s_f).$$

$\mathrm{RGB_{t3D}}$ denotes the color components (note that κ is a vector), and α_{t3D} the opacity of an entry in the 3D texture map. $\mathrm{RGB_{t3D}}$ and α_{t3D} depend on the texture coordinates l, s_f, s_b, and the transfer functions κ and ρ. Thus, the texture map has to be updated whenever the transfer functions are modified.

It is an intrinsic limitation of our method that κ and ρ have to depend on the same scalar field. However, we are not limited to this optical model; for example the model of Wilhelms and Van Gelder [68, 116, 120] could be implemented by simply replacing $\kappa(s_l(t))\rho(s_l(t))$ by a differential color vector $E(s_l(t))$ (or $g(s_l(t))$ in the notation of [68, 120]).

After the calculation of the texture map in a preprocessing step, all tetrahedra are projected from back to front. Before rendering the triangles of one projected tetrahedron, the three texture coordinates are set for each vertex of the triangles. Then they are blended appropriately into the frame buffer.

The blending operation corresponds to

$$I' = \mathrm{RGB}_{\mathrm{t3D}} + (1 - \alpha_{\mathrm{t3D}}) \times I,$$

and is done very efficiently by today's graphics hardware. We give a synthetic example of this rendering method in Figure 8.3. The scalar values at the vertices of the visualized tetrahedral mesh are determined by the distance of each vertex to the surface of a sphere. The transfer function of the opacity is 0 except for a small interval, which results in the two partially opaque rings in Figure 8.3.

In summary our method allows us to exploit hardware-supported 3D texture mapping in order to render projected tetrahedra without the need to do any time consuming calculations for each pixel. Our approach is not as accurate as ray-casting algorithms or the high accuracy (HIAC) volume rendering system described in [120] because of limited texture memory and non-linear transformations in the case of perspective projections. Especially limited texture memory will limit the size of the 3D texture map resulting in a less accurate resampling of the transfer functions. Within this limited accuracy, however, arbitrary transfer functions may be used without affecting the rendering times, whereas the performance of the HIAC system depends crucially on the chosen transfer functions. In particular, thin peaks are possible within our approach resulting in unshaded isosurfaces as demonstrated in Section 8.13.

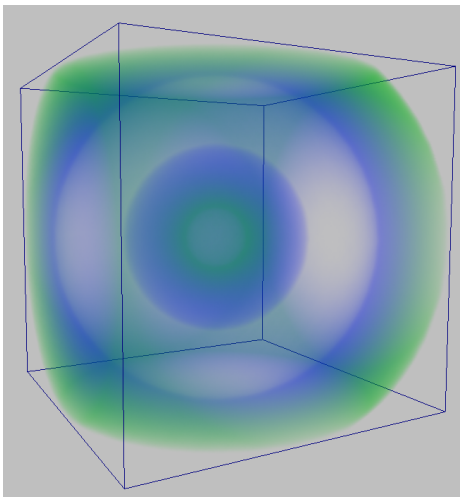


Figure 8.3: Visualization of a synthetic data set with non-linear transfer functions implemented with a 3D texture map of dimensions $64 \times 64 \times 64$ (1 MB).

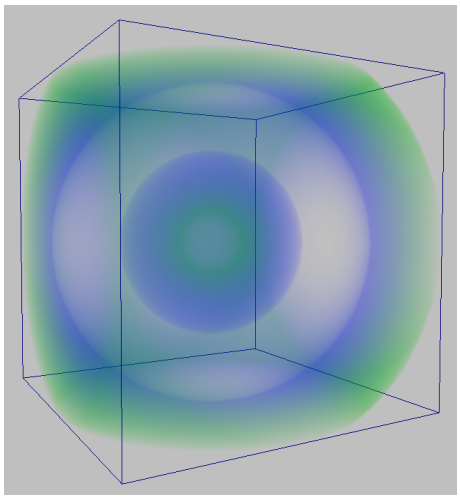


Figure 8.4: Same data set as in Figure 8.3 but rendered using a 2D texture map of dimensions 256×256 (256 KB). (See Section 8.5.)

8.5 A New Approximation for PT

As hardware-supported 3D texture mapping is not available on every graphics workstation, and the 3D texture maps that are employed in Section 8.4 need rather much texture memory, we will describe a new approximation to the rendering of projected tetrahedra using 2D texture mapping, which interpolates the opacity linearly. However, this method allows us to use arbitrary transfer functions, while existing hardware-accelerated solutions are limited to linear transfer functions within each cell (e.g. [106]).

The basic idea is to approximate the dependencies of the integrals in Equation (8.1) on l by linear terms, and to implement these terms by a modulation of the vertex colors. The remaining integrals depend only on s_f and s_b, and can thus be tabulated in a 2D texture map.

The dependencies on l in Equation (8.1) become more explicit with the variable substitutions $t' = t/l$ and $u' = u/l$:

$$\begin{aligned} I' &= l\int_0^1 \exp\left(-l\int_0^{t'} \rho(s_1(u'))\mathrm{d}u'\right) \\ &\quad \times \kappa(s_1(t'))\rho(s_1(t'))\mathrm{d}t' \\ &\quad + \exp\left(-l\int_0^1 \rho(s_1(t'))\mathrm{d}t'\right) \times I. \end{aligned}$$

For $l = 0$ this equation reduces to $I' = I$. For given ρ, κ, s_f and s_b we evaluate the integrals for another value $l = \bar{l} = \text{const.}$ and extrapolate linearly in l. The optimal choice of $\bar{l}$ depends on the particular application but setting $\bar{l}$ equal to the average cell thickness has proven to be a good approximation. The 2D texture map is defined by

$$\begin{aligned} \mathrm{RGB}_{\mathrm{t2D}} &= \bar{l}\int_0^1 \exp\left(-\bar{l}\int_0^{t'} \rho(s_1(u'))\mathrm{d}u'\right) \\ &\quad \times \kappa(s_1(t'))\rho(s_1(t'))\mathrm{d}t', \\ \alpha_{\mathrm{t2D}} &= 1 - \exp\left(-\bar{l}\int_0^1 \rho(s_1(t'))\mathrm{d}t'\right) \end{aligned} \tag{8.2}$$

and is modulated by colors at the vertices with the RGBα components set equal to $(l/\bar{l}, l/\bar{l}, l/\bar{l}, l/\bar{l})$. In practice we are scaling these colors by the maximum opacity value in the texture map in order to avoid clamping for values $l > \bar{l}$. This scaling is compensated by multiplying the entries in the texture map with the reciprocal value. The combined effect of texturing and blending with appropriate blending coefficients is

$$I' = \frac{l}{\bar{l}} \times \mathrm{RGB}_{\mathrm{t2D}} + \left(1 - \frac{l}{\bar{l}} \times \alpha_{\mathrm{t2D}}\right) \times I,$$

which is our new approximation of Equation (8.1). Accordingly, we use `GL_ONE` for the source blend factor and `GL_ONE_MINUS_SRC_ALPHA` for the destination blend factor in OpenGL.

On the one hand, this approximation results in artifacts because of the linear interpolation (see [106]), on the other hand, the use of 2D texture mapping enables us to utilize larger texture maps compared with the 3D texture maps employed in Section 8.4 resulting in an improved resampling of the transfer functions.

Figure 8.4 shows the synthetic example from Figure 8.3 using 2D instead of 3D texture mapping. The linear approximation results in slightly smaller opacities resulting in lighter colors, while the improved resampling results in sharper edges of the structures generated by the transfer functions. The middle image in Figure 8.12 represents an example of a 2D texture map generated by Equation (8.2).

The following sections discuss an independent extension of the PT algorithm capable of displaying smoothly shaded isosurfaces without vertex interpolations. Additionally, two methods are presented to combine projected tetrahedra with opaque isosurfaces.

8.6 Prior Work about Isosurfaces

For an in-depth introduction into current research about isosurfaces the reader is referred to [5]. Isosurfaces are an indispensable tool in volume visualization, although direct volume rendering includes much more information in one picture. However, isosurfaces are preferred for many applications as they are usually more comprehensible. Thus, direct volume rendering techniques are often extended with isosurfaces in order to combine the advantages of both techniques.

In their description of the PT algorithm [101] Shirley and Tuchman suggested to calculate isosurfaces based on a marching tetrahedra algorithm similar to the marching cubes algorithm [62, 63, 125]. The combination of these algorithms makes it possible to render isosurfaces with any degree of transparency as noted in [101].

However, research on marching cells algorithms concentrated on reducing the number of cells tested for intersections with the isosurface [6, 9, 10, 61, 100, 99, 117] and on simplifying the polygonal mesh representing the isosurface [27, 60, 76, 79, 98]. Instead of reducing the number of polygons point-based algorithms for the extraction of isosurfaces [20, 58, 81, 108] do not produce any polygons. Westermann's multi-pass algorithm for shaded isosurfaces [113] also does not construct a polygonal representation of the isosurface. As our algorithm is based on the same idea, we present the common concept in Section 8.7 before discussing our algorithm in Section 8.8.

8.7 Hardware-Accelerated Marching Cells

This section discusses a variant of the first pass of Westermann's algorithm for shaded isosurfaces in unstructured grids [113]. The algorithm presented here sets all pixels of the silhouette of an intersection of an isosurface with a tetrahedral cell. Figure 8.5a shows the resulting silhouette, while Figures 8.5b and 8.5c show intermediate steps of the algorithm.

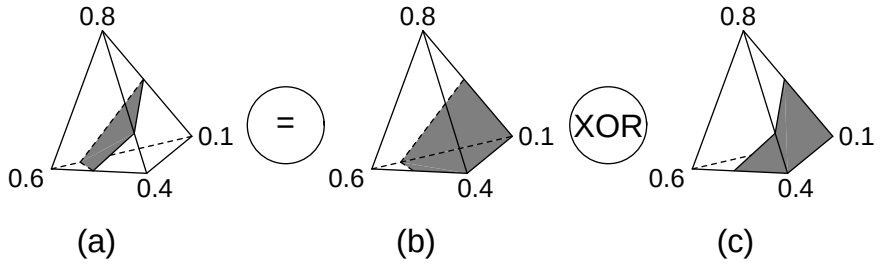


Figure 8.5: The polygon of an isosurface (isovalue 0.5) within a tetrahedral cell (a) can be obtained by an XOR combination of the two pictures (b) and (c). (b) shows the parts of the back faces of the cell with scalar value less than 0.5. (c) is the analogue to (b) for the front faces.

The first step is to render those parts of the back faces of the cell where the interpolated scalar value is less than the isovalue (see Figure 8.5b). Utilizing OpenGL this can be achieved by setting the α-components of the vertices' color to the scalar values and activating an appropriate α-test. Then the front faces are rendered in exactly the same way, i.e. again only those parts are rendered where the interpolated scalar value is less than the isovalue (see Figure 8.5c). By combining both pictures with an exclusive-OR (XOR) operation the correct set of pixels is obtained. Using OpenGL an XOR operation can be realized with the help of a 1-bit stencil buffer by inverting its contents whenever a pixel passes the α-test.

Note that the result is not sensitive to the order of the polygon rendering, i.e. the back and front faces could be rendered in any order. The result is also the same if the α-test is inverted for all faces, i.e. if those parts of the polygons are rendered where the interpolated scalar value is greater than the isovalue. Westermann's original algorithm differs in so far as the α-test is inverted for the back faces only and the pictures are combined with an AND-operation. However, this requires additional passes in order to generate both faces of the isosurface.

In summary this algorithm requires the rendering of all front and back faces in order to set the stencil buffer and to render either the front or the back faces once

more for flat-shaded isosurfaces. Thus, for a tetrahedral cell five to seven triangles have to be rendered, while a polygonal representation of the isosurface in a tetrahedron needs only one or two triangles. Therefore, the advantage of interpolating the scalar data with the help of OpenGL hardware is more than compensated by the need to render additional polygons.

The situation is, however, fundamentally different in the context of the PT algorithm as will be discussed in the following section.

8.8 Flat-Shaded Isosurfaces

As mentioned in Section 8.2 the PT algorithm [101] triangulates the projection of tetrahedra as shown in Figure 8.1. However, instead of referring to a triangulation of the projected silhouette into triangles, we can as well think of a decomposition of the original tetrahedron into smaller tetrahedra, which are projected after the decomposition. The projections of these smaller tetrahedra are all of the same kind: Two faces are degenerate and the other two faces are projected onto the same (non-degenerate) triangle. This observation enables us to reduce the algorithm presented in Section 8.7 to a single-pass algorithm for these tetrahedra using 2D texture mapping.

As explained in Section 8.7 pixels are set if and only if the interpolated scalar value of either the back or the front face is less than the isovalue. As noted the back and front face are projected onto the same triangle. Therefore, it is sufficient to render this triangle using a checkerboard-like, two-dimensional texture map as shown in the right-hand column of Figure 8.6 with the two texture coordinates corresponding to the interpolated scalar value of the back and front face, respectively. (See Section 8.13 for an alternative derivation of this 2D texture map.)

The first texture coordinate corresponds to the scalar value on the front face and the second texture coordinate to the scalar value on the back face. As the scalar data are interpolated linearly, the texture coordinates should also be interpolated linearly. Perspective corrections of texture coordinates should, therefore, be disabled. Actual values of texture coordinates have to be specified at the vertices of the triangle and are determined by the scalar data defined at the vertices of the projected tetrahedron. (See the left-hand column in Figure 8.6 for the scalar data defined at the vertices of the tetrahedron and the middle column for the resulting pairs of texture coordinates at the vertices of the projected triangle.) If the scalar data are not in the appropriate range for texture coordinates, the values have to be scaled accordingly. However, this can be done in a preprocessing step. The texture itself has to determine the α-component, i.e. the opacity, which has to be 1 for opaque isosurfaces whenever either the first or the second texture coordinate is less than the isovalue, and 0 otherwise (see the right-hand column of Figure 8.6).

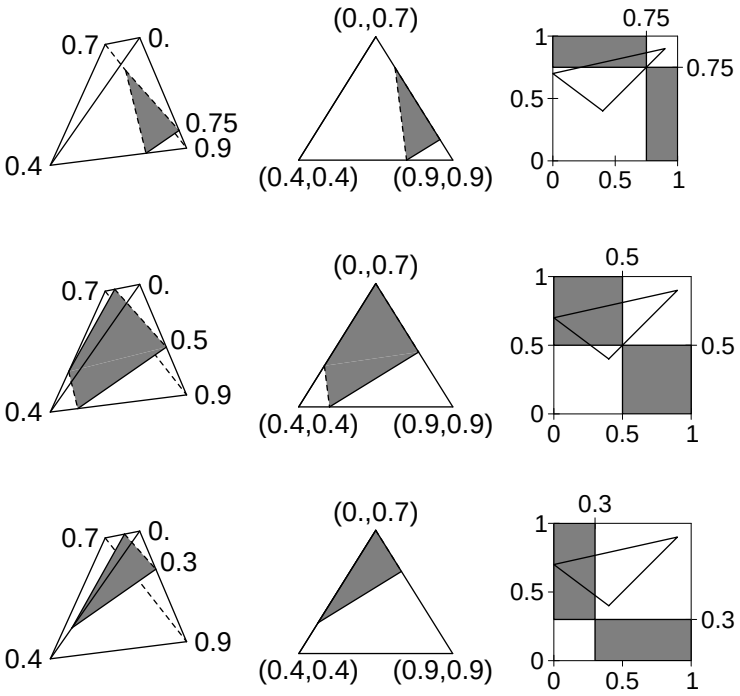


Figure 8.6: Projected tetrahedra (middle column) with flat-shaded isosurfaces for isovalues 0.75 (top row), 0.5 (middle row), and 0.3 (bottom row). The left-hand column shows the same tetrahedra slightly rotated with scalar data at the vertices. These values define the texture coordinates included in the pictures of the actual projections in the middle column. The right-hand column shows the corresponding texture maps including the triangles in the space of texture coordinates.

As this pass does not allow any kind of smooth shading, we employ flat shading, i.e. the RGB-components of the color of the triangle are constant.

Unfortunately, edges of isosurface patches within triangles (see the middle column of Figure 8.6 for some examples) will cause rendering artifacts as there is no mechanism which aligns them exactly to the corresponding edges in the projected tetrahedra in front or behind. We can avoid gaps by slightly modifying the texture map, effectively “thickening” the isosurface. This eliminates artifacts for opaque isosurfaces; for partially transparent isosurfaces, however, this will visually enhance edges of the tetrahedral mesh by rendering pixels twice. Removing these artifacts for partially transparent isosurfaces is an open problem and requires additional efforts in the future.

8.9 Smoothly Shaded Isosurfaces

Our algorithm for smoothly shaded isosurfaces is again a variant of the corresponding passes of Westermann's algorithm for shaded isosurfaces in unstructured grids [113]; however, there are several crucial differences. For each triangle the steps of our algorithm are:

1. Render the shaded back face triangle restricted to the isosurface silhouette as discussed in Section 8.8.

2. Repeat the preceding step for the front face triangle.

3. Form the weighted sum of the two pictures to get shading for intermediate positions of the isosurface.

The weights differ for each pixel as they depend on the relative distances of the isosurface to the front and back face, respectively (see Figure 8.7). For reasons which will become clear in the next paragraph, let α denote the weight of a pixel of the front triangle. According to Figure 8.7 the weight α is

$$\alpha = \frac{s_{\text{iso}} - s_b}{s_f - s_b} \quad \text{for} \quad s_f < s_{\text{iso}} < s_b \quad \text{or} \quad s_f > s_{\text{iso}} > s_b$$

with the isovalue s_{iso}; s_f and s_b were defined in Section 8.4. The weight of a corresponding pixel on the back face triangle is $1 - \alpha$. While weights for all pixels were calculated in software in [113], we are calculating the weighted sum completely in hardware.

We still use the 2D texture map of Section 8.8 for the back face triangle but employ a modified version of this texture map for the front face triangle. This new texture map (see Figure 8.8 for an example) is modulated with the weights α. As the original texture map contains only opacity values 0 and 1, this modulated map in fact stores the weights $\alpha = \frac{s_{\text{iso}} - s_b}{s_f - s_b}$ for the front face triangle. (Remember that s_f and s_b are the texture coordinates and that the texture map already depends on s_{iso}.) Thus, the weights α in fact specify opacities. Using this texture map when rendering the front face triangle and blending it appropriately onto the opaque back face triangle generates, therefore, the correct weighted sum of both triangles.

Thus, our algorithm for smoothly shaded isosurfaces can be reformulated in two passes for each tetrahedron:

1. Render the shaded back face triangle restricted to the isosurface silhouette. (See Section 8.8.)

2. Blend the shaded front face triangle modulated with a texture map containing the correct weights onto the back face triangle.

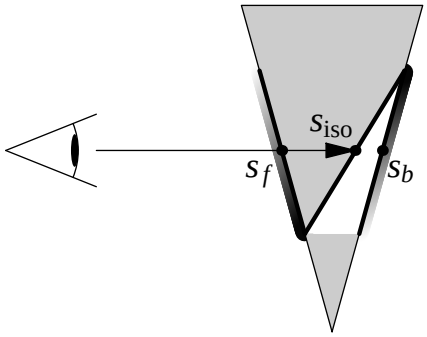


Figure 8.7: Rendering smoothly shaded isosurfaces by shading the back and front face triangle, and forming the weighted sum. Weights are symbolized by gray scales and are determined by the relative distances of the front and back faces to the isosurface given by $(s_{\text{iso}} - s_b)/(s_f - s_b)$ and $(s_{\text{iso}} - s_f)/(s_b - s_f)$ respectively.

Special care has to be taken with vertices from the decomposition of projected tetrahedra, because they can result in artifacts similar to those induced by hanging nodes. Therefore, the color of a vertex inserted between two vertices of the mesh has to be equal to the color generated by the graphics hardware interpolating between these vertices.

The algorithm was used in Figure 8.10 to render several isosurfaces of different colors as explained in the following section.

8.10 Colored and Multiple Isosurfaces

The techniques presented in Sections 8.8 and 8.9 can be extended to colored and multiple isosurfaces. Coloring can be achieved by setting the vertex colors to white and modulating them with colored RGBα texture maps. The two faces of an isosurface can be colored independently by choosing different colors for texels with $s_f > s_b$ and $s_f < s_b$ respectively.

An example of a texture map for multiple isosurfaces is given in Figure 8.9, which shows the combination of the (colored) texture maps from Figure 8.6. The "visibility ordering" is easy to understand: For $s_f < s_b$ we view along the gradient of the scalar field, thus isosurfaces for smaller isovalues occlude those for greater isovalues, and vice versa for $s_f > s_b$.

Assuming that all cells are rendered, the number of isosurfaces n in the texture map does not affect the rendering time. For opaque isosurfaces our method shares this feature with Westermann's algorithm for multiple isosurfaces [114], while

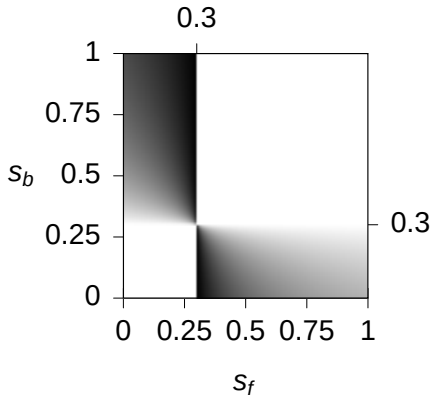


Figure 8.8: A 2D texture map used for a front face triangle; black corresponds to opacity 1 (opaque), white to opacity 0 (transparent). It is a modulation of the lower texture map in Figure 8.6 with the weights $\alpha = \frac{s_{\text{iso}} - s_b}{s_f - s_b}$ and $s_{\text{iso}} = 0.3$.

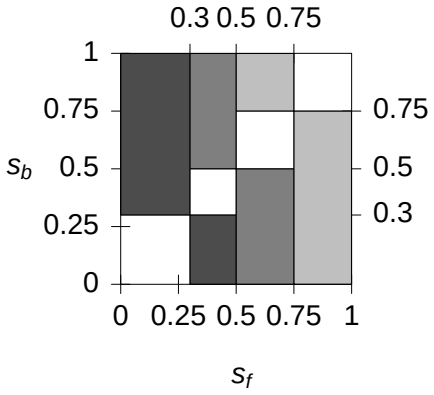


Figure 8.9: The correct combination of the texture maps from Figure 8.6 into a single texture map for multiple isosurfaces. (See Section 8.10.)

ray-casting approaches depend at least logarithmically on n. For partially transparent isosurfaces our method does still not depend on n while the dependency of ray-casting approaches changes to n.

8.11 Mixing Isosurfaces with Projected Volumes

It was claimed that rendering mixtures of opaque polygons and volumetric data is straightforward, e.g. in [52]. This claim, however, does not apply to any cell projecting approach including the PT algorithm, since special attention has to be paid to partially occluded cells. In [120] Williams et al. suggest to slice each cell at user-specified isovalues. The time complexity of this method, however, depends linearly on the number of isosurfaces. As we noted in Section 8.10 the time complexity of our algorithm does not depend on the number of isosurfaces; therefore, we propose two alternative methods of mixing isosurfaces and volumes, which are more appropriate in this context.

The algorithm presented in Section 8.9 allows us to smoothly include projected tetrahedra by rendering them after the corresponding back face triangle and before the front face triangle. This order ensures that the projected volume is completely occluded where the front face triangle is opaque, i.e. where the isosur-

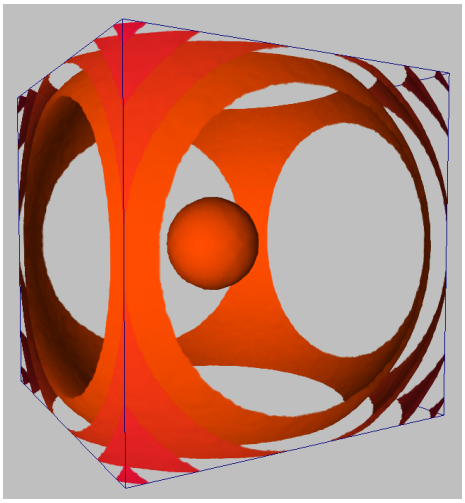


Figure 8.10: Several isosurfaces extracted from the data set shown in Figures 8.3 and 8.4.

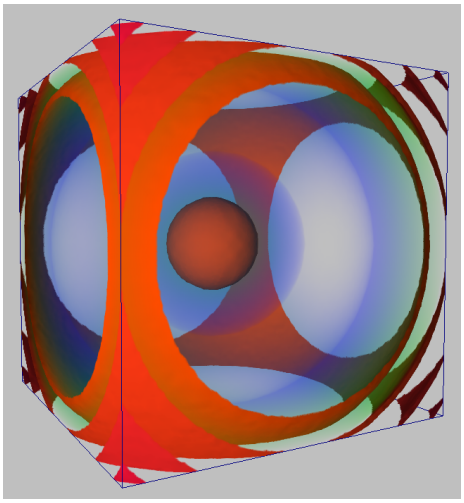


Figure 8.11: Smooth combination of Figures 8.4 and 8.10. (See Section 8.11.)

face is in front of the volume at the front face, and that the volume is not affected where the front face is transparent, i.e. where the isosurface is behind the volume at the back face. Figure 8.7 illustrates this correlation: The relative thickness of the occluded part of the tetrahedron (white) corresponds to the weight of the front face (left gray scale).

An example employing this method is given in Figure 8.11, which mixes the isosurfaces of Figure 8.10 with the projected tetrahedra of Figure 8.4. More realistic examples are presented in Figures 8.13, 8.14, and 8.15. Figure 8.12 comprises the three 2D texture maps required to render the NASA Bluntfin data set (Figure 8.13).

Although our approach avoids discontinuities, it is not completely accurate with respect to correct ray integration. Therefore, we developed a more rigorous method. For opaque isosurfaces the ray integration in Equation (8.2), respectively Equation (8.1) if 3D texture mapping is employed, has to be stopped as soon as one of the isovalues is reached, i.e. for $s_l(t) = s_{\text{iso}}$ (see Figure 8.7). By rendering the isosurfaces for each triangle first (either in one pass for flat-shaded isosurfaces or two passes for smoothly shaded isosurfaces), followed by the projected volume with the modified 2D or 3D texture map, we are able to generate an accurate picture.

An example of a 2D texture map generated this way is shown in the middle image of Figure 8.12. The isosurfaces manifest themselves in transparent vertical stripes which correspond to a scalar value s_f on the front face of a tetrahedron

slightly greater than one of the isovalues. Both methods presented in this section can be generalized to partially transparent isosurfaces.

8.12 Performance Comparison

With hardware-accelerated texture mapping the direct volume rendering methods presented in Sections 8.4 and 8.5 are essentially as fast as existing implementations of the PT algorithm. We emphasize that the rendering times for our methods are not affected by the particular transfer functions employed.

Our extensions of the PT algorithm are hard to compare with "non-PT" algorithms for direct volume rendering, e.g. approaches based on slicing, because the most time critical step of the PT algorithm is the sorting of the tetrahedra, which is not affected by the extensions presented in this chapter.

The algorithms for the rendering of isosurfaces described in Sections 8.8 and 8.9 depend on the correct sorting and decomposition of the tetrahedral cells, while most of the algorithms mentioned in Section 8.6 do not require any sorting or decomposition of tetrahedra. Moreover, we did not attempt to reduce the number of cells tested for intersections with the isosurface. Thus, most of the algorithms mentioned in Section 8.6 will usually be faster than our current implementation if used to render only a single isosurface. However, as our worst-case rendering time does not depend on the number of isosurfaces, our method will outrun most of the other algorithms if the number of isosurfaces is large enough (see also Table 8.1).

Moreover, our rendering algorithms greatly benefit from a combination with projected volume cells as described in Section 8.11 because the sorting and decomposition of tetrahedra can be reused in this scenario. Thus, the inclusion of isosurfaces in a visualization application based on the PT algorithm is almost for free. As the rendering in our methods includes extraction and triangulation of the isosurface, the rendering time (without sorting and decomposition of tetrahedra) should be compared to the sum of the extraction, triangulation, and rendering times of other algorithms. Additional efforts required by other algorithms for partially transparent isosurfaces and mixing with volume cells should also be considered in a fair comparison.

The rendering times in Table 8.1 were obtained on an Octane MXE with a MIPS R10K 250 MHz CPU. The isosurfaces were extracted from the NASA Bluntfin data set, which was converted into 187,395 tetrahedra. An image with three isosurfaces is depicted in Figure 8.13 and the corresponding pre-integrated texture maps are visualized in Figure 8.12. Clearly the rendering times for flat-shaded isosurface depend on the number of intersected tetrahedra (no double-counting) instead of the number of isosurfaces. Smoothly shaded isosurfaces require about twice as much time because the back and front faces have to be

no. isosurfaces	no. cells	flat-shaded	smoothly shaded
1	14,729	0.09 sec.	0.22 sec.
2	25,361	0.20 sec.	0.41 sec.
10	25,361	0.20 sec.	0.41 sec.

Table 8.1: Rendering times (including "extraction" and "triangulation") for isosurfaces from the NASA Bluntfin data set. The number of cells refers to the number of tetrahedra intersected by at least one isosurface. Timings for the sorting and decomposition of tetrahedra are not included as these steps are already done by the original PT algorithm without our extensions.

rendered separately. For a single, smoothly shaded isosurface our rendering time is close to the 0.2 seconds reported by Westermann in [113]. The rendering performance is comparable to the results of Wittenbrink in [121].

8.13 Heaviside Excursion

This section demonstrates the extraction of unshaded isosurfaces with the technique presented in Section 8.4 by choosing an appropriate transfer function ρ. As a side effect the 2D texture maps of Section 8.8 reveal themselves as special cases of the 3D texture map of Section 8.4. In order to extract the isosurface for an isovalue s_{iso} we have to set $\rho(s) = 0$ for $s \neq s_{\text{iso}}$ and $\rho(s_{\text{iso}}) = \infty$. Formally, we set $\rho(s) = C\delta(s - s_{\text{iso}})$ with a large constant C and Dirac's delta function $\delta(x)$; multiple isosurfaces correspond to a sum of delta functions. As $\kappa(s_{\text{iso}})$ is constant, we are only interested in the value of α as defined in Equation (8.1):

$$\begin{aligned}
1-\alpha &= \exp\left(-\int_0^l \rho(s_l(t))\mathrm{d}t\right) \\
&= \exp\left(-\int_0^l C\delta(s_f + \frac{t}{l}(s_b - s_f) - s_{\text{iso}})\mathrm{d}t\right) \\
&= \exp\left(-\int_0^l C\left|\frac{l}{s_b - s_f}\right|\delta\left(t - l\frac{s_{\text{iso}} - s_f}{s_b - s_f}\right)\mathrm{d}t\right) \\
&= \exp\left(-C'H\left(\frac{s_{\text{iso}} - s_f}{s_b - s_f}\right)H\left(\frac{s_{\text{iso}} - s_b}{s_f - s_b}\right)\right)
\end{aligned}$$

Let $C' = C\left|\frac{l}{s_b - s_f}\right|$ and let $H(x)$ be the Heaviside step function (see [21]). Thus, for $C \to \infty$ we obtain

$$\alpha = H\left(\frac{s_{\text{iso}} - s_f}{s_b - s_f}\right)H\left(\frac{s_{\text{iso}} - s_b}{s_f - s_b}\right),$$

which is independent of l. The dependency on s_f and s_b is already visualized in the texture maps shown in Figure 8.6. Obviously, the 2D texture maps used in Section 8.8 are in fact special cases of the 3D texture map of Section 8.4. However, the derivation presented in Sections 8.7 and 8.8 appears to be more intuitive and comprehensible.

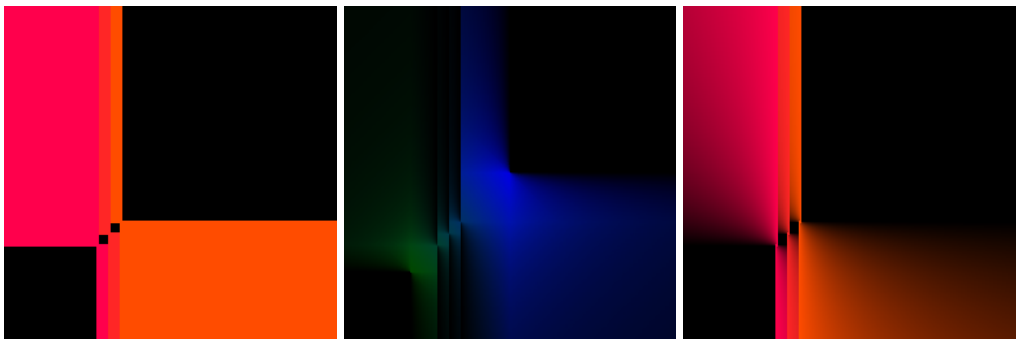


Figure 8.12: These 2D texture maps of dimensions 256×256 were used to render the Bluntfin data set depicted in Figure 8.13. The left and right textures were employed to render the back and front face triangles, whereas the projected volume was generated by the middle texture with pre-integration stopped at the isovalues. The texels on the diagonal of this texture represent the transfer functions. Black pixels in these images correspond to completely transparent texels.

8.14 Pre- vs. Post-Classification

The original approach of Shirley and Tuchman is often called pre-classification, because the transfer functions are applied to the scalar values of each cell before rendering. In contrast to this, our cell projection approach is called post-classification, since emission and opacities are derived after cell projection. Post-classification is able to reproduce the ray integral accurately, which means that the influence of the transfer function can be reproduced accurately inside the tetrahedra. Pre-classification simply neglects the non-linear contributions of the transfer function. This becomes especially visible if the transfer function contains a discontinuity. This is best illustrated by a direct comparison of the methods as shown in Figure 8.16.

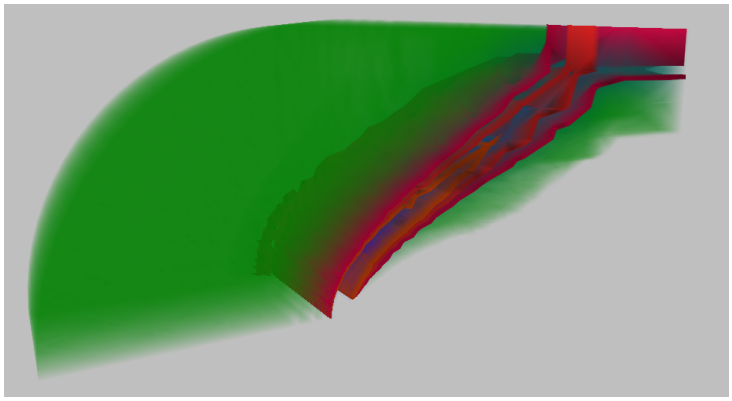


Figure 8.13: Visualization of the Bluntfin data set with three isosurfaces mixed with projected tetrahedra.

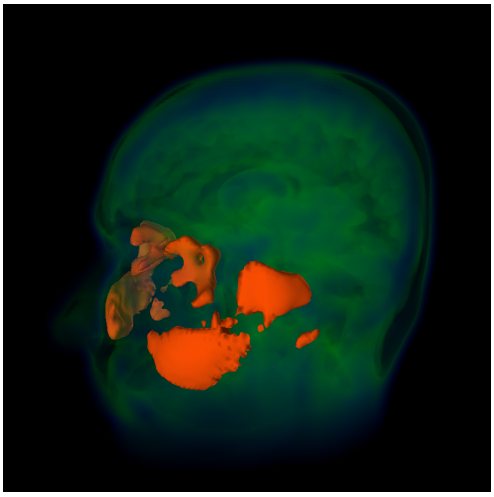


Figure 8.14: A visualization of an MRI head scan. The orange isosurface depicts soft tissue located in the cheeks and behind the eye balls.

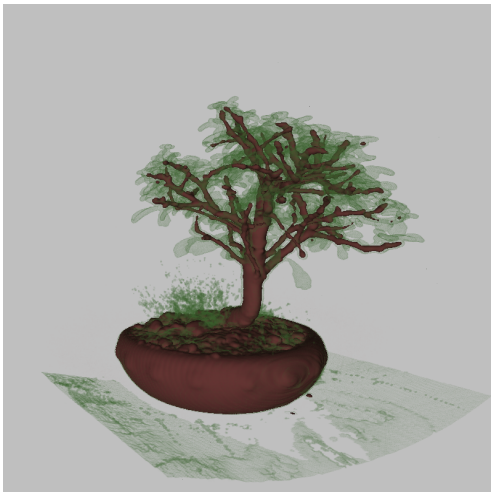


Figure 8.15: A CT scan of a bonsai: Leaves are visualized by direct volume rendering, while the trunk and the branches are shown by the brown isosurface.

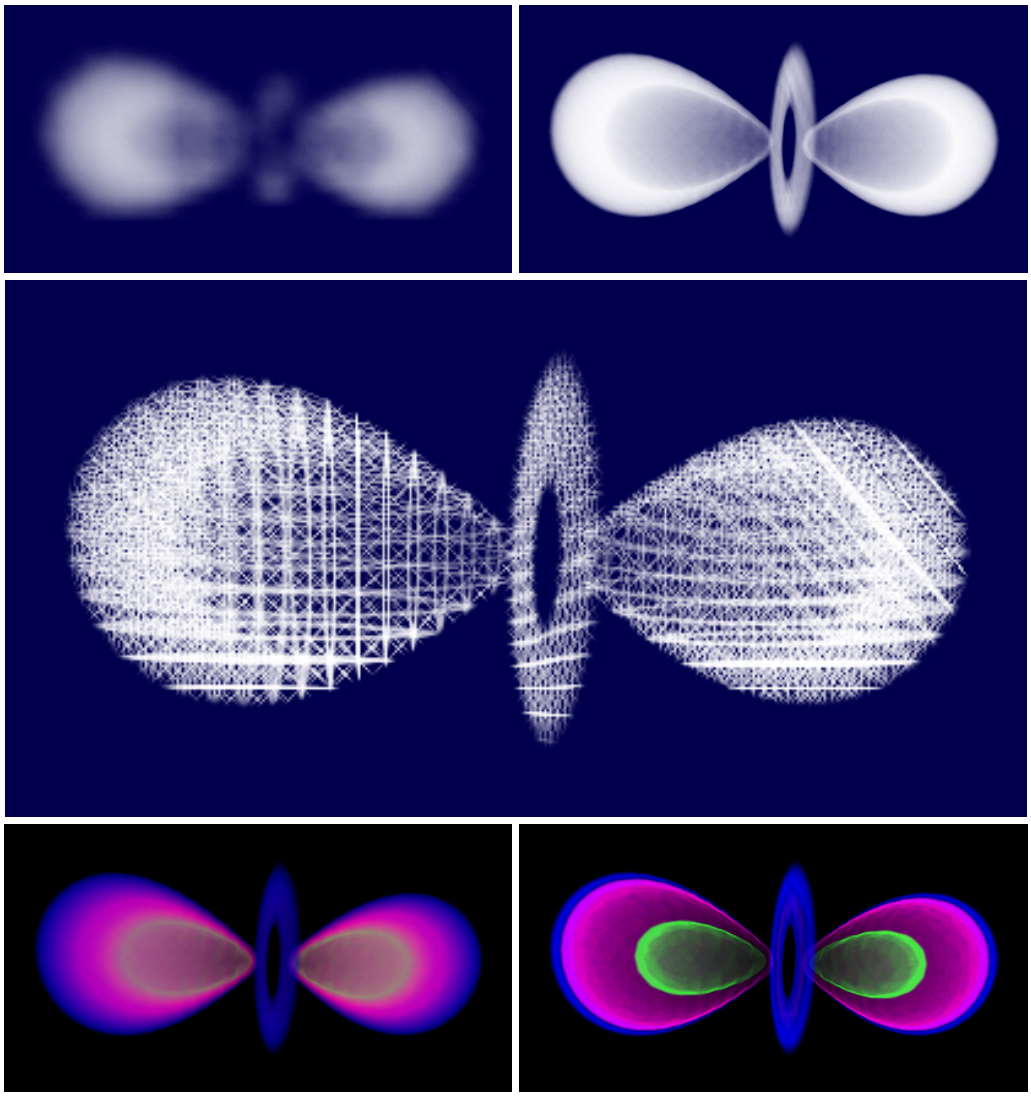


Figure 8.16: Pre- vs. post-classification: The example data set shows the electron density of a hydrogen atom. **Top row**: pre-classification (left) and post-classification (right) with a transfer function that cuts away low electron densities. Whereas the sharp cut is smoothed out on the left, on the right the cut is reproduced correctly inside the tetrahedra. **Center**: wire-frame view. **Bottom row**: post-classification with color ramp as transfer function, and the extraction of three unshaded isosurfaces by setting the transfer function to thin differently colored peaks.

Chapter 9

Unstructured Volume Rendering on the PC

The presented cell projections method significantly enhances the quality of unstructured volume rendering, but not yet fully exploits the capabilities of current PC graphics accelerators. Due to the increasing flexibility of commodity graphics hardware the pre-integration technique has become widely available for the visualization of volume data on regular grids. The previous approach for unstructured meshes employed a 3D texture to effectively apply pre-integration. Although the resulting images are of high quality, there are several restrictions due to the limited amount of available texture memory. Transfer functions with high gradients require a high resolution pre-integration table, which does not fit easily into the dedicated texture memory. Modern PC graphics hardware, for instance the ATI Radeon 8500 and the NVIDIA GeForce4, allow more sophisticated approaches using dependent textures, multi-texturing, per-pixel shading, and hardware accelerated pre-integration. This enables us to overcome the limited size of the three-dimensional pre-integration table. To circumvent this restriction we propose to implicitly store the 3D texture by means of multiple 2D textures. Then the colors and opacities of the three-dimensional pre-integration table can be reconstructed accurately with the high internal precision of the pixel shader.

9.1 High Resolution Ray Integral

Since high resolution 3D textures require huge amounts of texture memory, we separate the three-dimensional function of the volume density optical model. Unfortunately, only the opacity can be separated easily. The chromaticity needs to be approximated by means of a linear combination of two-dimensional functions.

9.1.1 Opacity Reconstruction

Since the opacity depends on the average density along the viewing ray and the length l of the ray segment, it can be separated as follows:

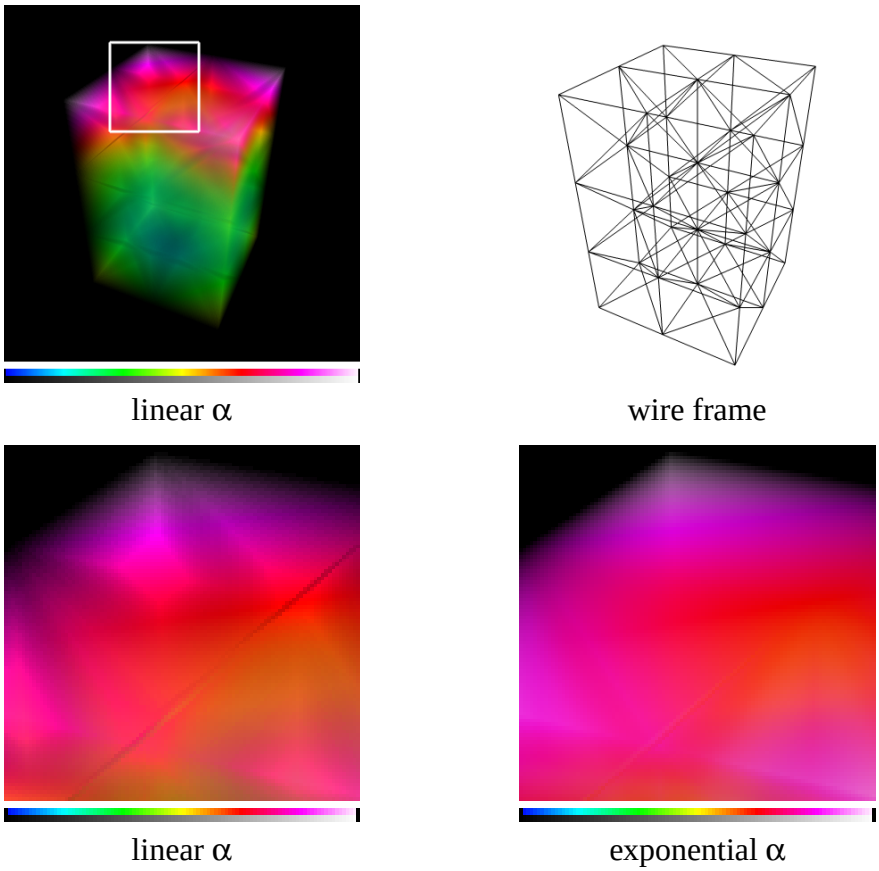


Figure 9.1: Comparison between linear approximation and correct exponential α. The corresponding transfer function which is split into normalized emission and opacity can be seen below each image.

$$\hat{\rho}(s_f, s_b) = \int_0^1 \rho(s_f + t(s_b - s_f))\mathrm{d}t \tag{9.1}$$

$$\alpha_{1D}(x) = 1 - e^{-x} \tag{9.2}$$

For each rendered pixel we derive the average density $\hat{\rho}$ from a 2D texture map (Equation 9.1) and compute the final opacity α_{1D} by means of a 1D dependent texture lookup (Equation 9.2).

In order to further increase the accuracy of the reconstructed α values, the dependent texture is extended to hold the higher 8 bits of a 16 bit α value in the alpha channel A and the lower 8 bits in the additional luminance channel L. In order to map the maximum 16 bit α value to 1, it is scaled by the factor $\frac{256}{257}$. Since the resulting equation $\alpha_{1D} = \frac{256A}{257} + \frac{L}{257}$ is linear, the texture interpolation delivers a true 16 bit α lookup.

unit	coordinates	RGB	A
0	s_f, s_b	$\hat{C}_0(s_f, s_b)$	$\rho(s_f, s_b)$
1	s_f, s_b	$\Delta C_1(s_f, s_b)$	-
2	$l\rho(s_f, s_b)$	-	$\alpha_{1D}(l\rho(s_f, s_b))$

Table 9.1: Texture setup for dependent texture mapping.

Compared to the linear approximation of the opacity using the 2D texturing approach as outlined in Section 8.5 the resulting images are significantly improved, as illustrated in Figure 9.1.

9.1.2 Chromaticity Reconstruction

In order to achieve a high-quality approximation of the chromaticity, we pre-integrate the normalized chromaticities $\hat{C}_l = \frac{C_l}{\alpha_l}$ for $l \to 0$ and $l = l_{max}$ with l_{max} being the maximum length of the ray segments. The normalized emission $\hat{C}_0$ and the difference ΔC_1 of the normalized emissions $\hat{C}_0$ and $\hat{C}_{l_{max}}$ are stored in two high resolution 2D textures. The latter emissions are defined as follows:

$$\begin{aligned}
\hat{C}_0(s_f, s_b) &= \frac{\int_0^1 \kappa(S_1(t))\rho(S_1(t))}{\int_0^1 \rho(S_1(t))} \\
\hat{C}_{l_{max}}(s_f, s_b) &= \frac{C(s_f, s_b, l_{max})}{\alpha(s_f, s_b, l_{max})} \\
\Delta C_1(s_f, s_b) &= \hat{C}_{l_{max}}(s_f, s_b) - \hat{C}_0(s_f, s_b)
\end{aligned}$$

Using the texture setup of Table 9.1, we implement the following approximation of the volume optical density model by utilizing dependent textures and the pixel shader on the NVIDIA GeForce4 [128] and the ATI Radeon 8500 [74] graphics adapter:

$$\begin{aligned}
C(s_f, s_b, l) &\approx \hat{C}_{lin}(s_f, s_b, l)\alpha(s_f, s_b, l) \\
\hat{C}_{lin}(s_f, s_b, l) &= \hat{C}_0(s_f, s_b) + \frac{l}{l_{max}}\Delta C_1(s_f, s_b) \\
\alpha(s_f, s_b, l) &= \alpha_{1D}(l\rho(s_f, s_b))
\end{aligned}$$

This is a linear approximation in l for every pair of s_f and s_b. As seen in Figure 9.2, the linear approximation is not accurate for transfer functions that contain high gradients. For an improved reconstruction we approximate the chromaticity

by a polynomial of degree $n > 1$ in l with the coefficients $\tilde{C}_i$, $i = 0 \ldots n$. This is similar to the polynomial texture mapping approach of Malzbender et al. [64], which reconstructs the colors of a surface by a biquadratic polynomial. In our case the approximated chromaticity is given by the polynomial

$$C(s_f, s_b, l) \approx \alpha(s_f, s_b, l) \sum_{i=0}^{n} \frac{l^i}{l_{max}^i} \tilde{C}_i(s_f, s_b).$$

To compute the polynomial coefficients $\tilde{C}_i$ we pre-integrate the chromaticity at $l = \frac{i}{l_{max}}$ for $i = 0 \ldots n$ and construct a polynomial through each of these points for every pair of s_f and s_b. This corresponds to the computation of $n+1$ slices with $l = const$ of the pre-integration table.

Since the number of texture units is limited, we can only use a polynomial approximation with a degree of up to 2 on the GeForce4 and of up to 4 on the Radeon 8500. In the latter case the rasterization performance drops by almost 50%, but the quality of the approximation is only improved slightly. Therefore a polynomial degree of 2 should be preferred (see Figure 9.2). The corresponding texture setups are depicted in Table 9.2. The polynomial coefficients are scaled to the maximum possible texel range $[-1 \ldots 1]$ to improve the precision of the approximation. Additionally, the α values are reconstructed with 16 bits of accuracy.

unit	coordinates	RGB	A
0	s_f, s_b	$\tilde{C}_0(s_f, s_b)$	$\rho(s_f, s_b)$
1	s_f, s_b	$\tilde{C}_1(s_f, s_b)$	-
...	...	...	...
n	s_f, s_b	$\tilde{C}_n(s_f, s_b)$	-
n+1	$l\rho(s_f, s_b)$	-	$\alpha_{1D}(l\rho(s_f, s_b))$

Table 9.2: Texture setup for polynomial color approximation of the three-dimensional ray integral (with a maximum polynomial degree of $n = 2$ on the GeForce4 and of $n = 4$ on the Radeon 8500).

9.2 Hardware Accelerated Pre-Integration on the ATI Radeon 8500

In order to visualize volume data comfortably one needs to change the transfer function interactively. Whenever the transfer function is modified the pre-integration table has to be recomputed. For a resolution of 512^2 and a polynomial

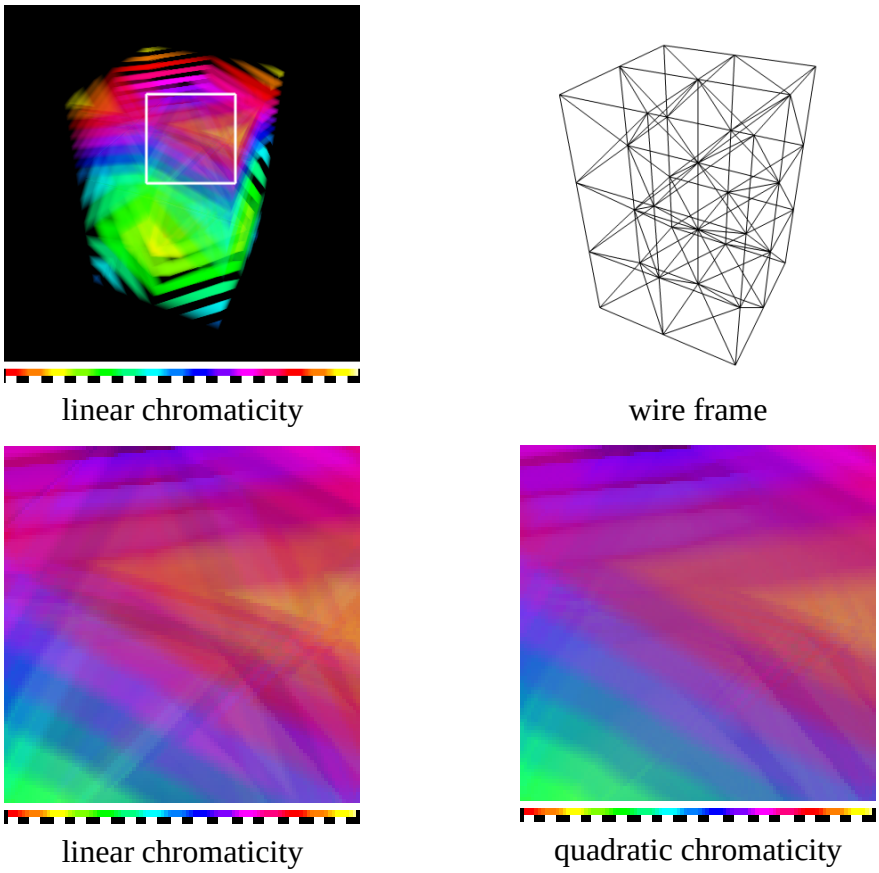


Figure 9.2: Comparison between linear and quadratic color approximation combined with 16 bit α, for the transfer function seen below each image.

degree of 4, for instance, this requires approximately 11 seconds on a Pentium 4 running at 2 GHz which is far too slow for interactive updates of the transfer function. In order to speed up the calculation of the pre-integration table we utilize graphics hardware for the purpose of numerical integration. We maintain a high level of accuracy by using the high internal precision of the pixel shader.

The numerical integration of the ray segments is performed by sampling the integral m times. At each sampling step, the integrated chromaticity κ and the integrated opacity α are blended with the corresponding entries of the transfer function.

As described by Engel et al.[26] the integrated opacity can be calculated quickly by the difference of two definite integrals. If self-attenuation is assumed to be negligible the same approach can be used to efficiently calculate the integrated chromaticities. This assumption is valid for volume slicing, since the ray segment

lengths l are usually small. In the case of unstructured volume rendering, however, this assumption does not hold, thus self-attenuation cannot be neglected. As a consequence, the numerical integration of the chromaticities is not fast enough to achieve interactive updates of the transfer function. However, the chromaticities of one slice of the pre-integration table can be integrated in parallel by using a hardware-accelerated approach. For each slice with a constant ray segment length l this is accomplished by blending m quadrilaterals containing the sampled transfer function for every pair of s_f and s_b into the frame buffer. The sampled transfer function is reconstructed from a 1D texture (see Table 9.3). For this purpose, the texture coordinate s of each vertex of the quadrilaterals is assigned as shown in Figure 9.3.

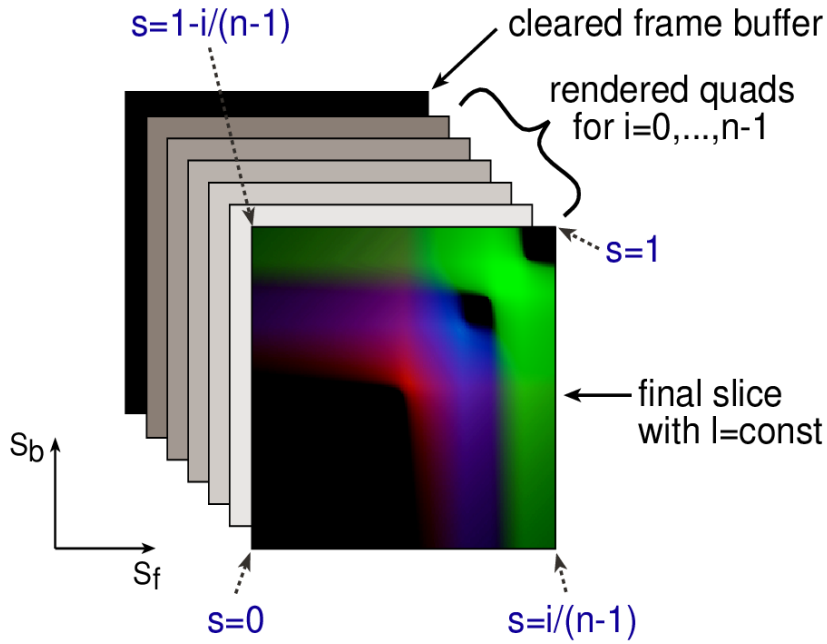


Figure 9.3: Hardware-accelerated pre-integration: First, the transfer function is stored in a 1D texture map. Then n rectangles are drawn and blended with the texture coordinate s assigned in the depicted fashion.

As the 8 bit frame buffer depth of current PC graphics hardware limits the accuracy of the numerical integration, we integrate the chromaticity with the higher internal accuracy of the pixel shader. Combining two channels of the frame buffer for each integrated color component of the chromaticity, a total accuracy of 16 bit can be achieved. In practice however, a bit depth of 12 has turned out to be sufficient.

We store the chromaticity and opacity of the transfer function for a given length l and the number of integration steps m in a 1D texture as defined in Table 9.3. To effectively represent high gradients in the transfer function, we construct the 1D texture with the highest possible resolution instead of using a linear interpolation of the 1D texture.

channel	meaning	value
red	high 8 bit (chromaticity)	$\kappa(s)$
green	low 4 bit (chromaticity)	$\kappa(s)$
blue	high 8 bit (opacity)	$1-e^{-\frac{l}{m}\rho(s)}$
alpha	low 4 bit (opacity)	$1-e^{-\frac{l}{m}\rho(s)}$

Table 9.3: 1D texture used for hardware-accelerated pre-integration.

On the Radeon 8500 the numerical integration is implemented using a method called ping-pong filtering [74]. For each blending step an RGBA texture contains the previously integrated chromaticity in the red (high 8 bits) and alpha channel (low 4 bits). First, the original 12 bit chromaticity is reconstructed in the pixel shader by multiplying the low bits with $\frac{1}{256}$ and adding the result to the high bits. Note that a texture entry of 255 in the high bits already represents a value of 1.0. Next, the chromaticity and opacity of the transfer function are reconstructed from the 1D texture in the same fashion. Then the chromaticity is multiplied by the opacity, the result of the previous iteration is multiplied by one minus the opacity, and the sum of both yields the new integrated chromaticity. Finally, the integrated chromaticity is split into 8 high and 4 low bits and is written back into the corresponding ping pong texture.

The Radeon 8500 masks out all bits representing values higher than 1.0 or lower than $\frac{1}{256}$. Therefore the high 8 bits are extracted automatically, whereas the low 4 bits are extracted by simply multiplying the 12 bit chromaticity with 256. In contrast to this, the GeForce4 always uses saturation logic instead of bit masking. Therefore the low 4 bits can only be extracted on the Radeon 8500. It should be noted that the described approach for high-accuracy blending is not necessary on the latest generation of PC graphics accelerators such as the ATI Radeon 9800 which allow floating point render targets.

A speedup of nearly 100% is achieved by performing four subsequent integration steps at once in the pixel shader. Since each RGB color component has to be computed separately, the hardware-accelerated pre-integration needs to be performed three times for every required slice of the pre-integration table. Each component of a pre-integrated slice is transferred back into main memory and recombined with the other color channels. This results in 9 pre-integration cycles for a polynomial approximation of a degree of 2, for example.

In contrast to software numerical integration, this hardware-accelerated approach allows to update the pre-integration table interactively. With respect to integration accuracy the hardware-accelerated method exhibits a higher integration error which is due to the 12 bit quantization. An example of these quantization artifacts is given in Figure 9.4.

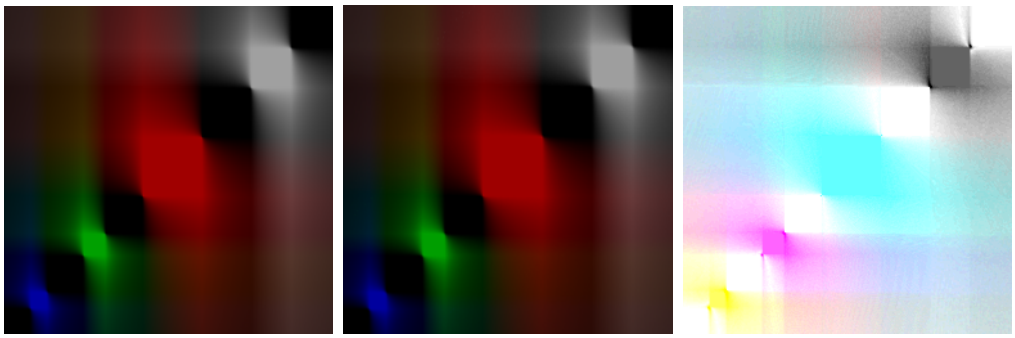


Figure 9.4: Comparison between hardware (left) and software (middle) pre-integration, including the error (right, scaled by a factor of 8 and inverted) for $m = 128$ sampling steps.

9.3 Performance Measurements

In the previous chapters we have demonstrated that the multi-texturing capabilities of modern PC graphics accelerators can be utilized to bring high-quality pre-integrated volume rendering of unstructured grids to the PC platform. Because of the reduced memory requirements of the employed 2D textures, our method is capable of applying high resolution transfer functions. A comparison of the visual quality of the proposed methods is given in Figure 9.5 together with visualizations of the Blunt Fin and the Bucky Ball data sets. The best approximation of the pre-integration table is achieved by using 16 bits for the representation of the opacities and a polynomial of degree 4 for the reconstruction of the chromaticities. A polynomial of degree 2 is only slightly less accurate, but performs significantly faster due to reduced rasterization requirements. Because of the high internal precision of the pixel shader and the representation of the opacities with 16 bits the results are even better than those obtained with a 3D texturing setup. Using our hardware-accelerated pre-integration approach we are able to maintain high update rates of the pre-integration table. In comparison to software integration the achieved speedup is about 700% on a PC equipped with a Pentium 4 running at 2 GHz and an ATI Radeon 8500 (compare Table 9.4).

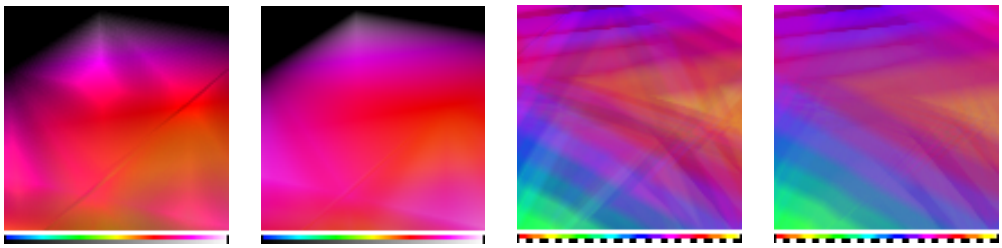
software	setup of textures
linear color ($n = 1$)	4.4s
polynomial $n = 2$	6.6s
polynomial $n = 4$	11.0s
Radeon 8500	setup of textures
linear color ($n = 1$)	0.6s
polynomial $n = 2$	1.0s
polynomial $n = 4$	1.7s

Table 9.4: Preprocessing times for 2D multi-texturing with a texture resolution of 512^2 texels.

The total rendering time is almost independent of the chosen reconstruction method (except for $n = 4$). It depends mainly on the sorting algorithm [118, 104, 121] and the transfer speed between the CPU and the graphics adapter (see Table 9.5). For comparison purposes the experimental results are given for a polynomial degree of 2. We achieve up to 600,000 tetrahedra per second depending on the sorting algorithm. Approximately half of the time is spent by sorting, while the other half is spent by rendering. The lower performance for rendering the Bucky Ball data set is due to a larger variation of the scalar values which lead to a reduced texture cache coherence.

GeForce4	#tetra	numeric	MPVO	XMPVO
Blunt Fin	187k	3.18 fps	2.64 fps	2.35 fps
Bucky Ball	177k	2.46 fps	2.19 fps	2.05 fps
Radeon 8500	#tetra	numeric	MPVO	XMPVO
Blunt Fin	187k	2.51 fps	2.20 fps	1.99 fps
Bucky Ball	177k	2.09 fps	1.98 fps	1.87 fps

Table 9.5: Display times including visibility sorting on a Pentium 4 running at 2 GHz using a polynomial approximation of degree 2 and a 1280×960 view port. The applied sorting algorithms are *numerical sorting* [121], *MPVO* [118], and *XMPVO* [104].



linear approximation of α | exponential interpolation of α | linear color reconstruction | quadratic color reconstruction

Figure 9.5: Comparison between different approximations of the ray integral. The applied transfer functions are depicted below each image.

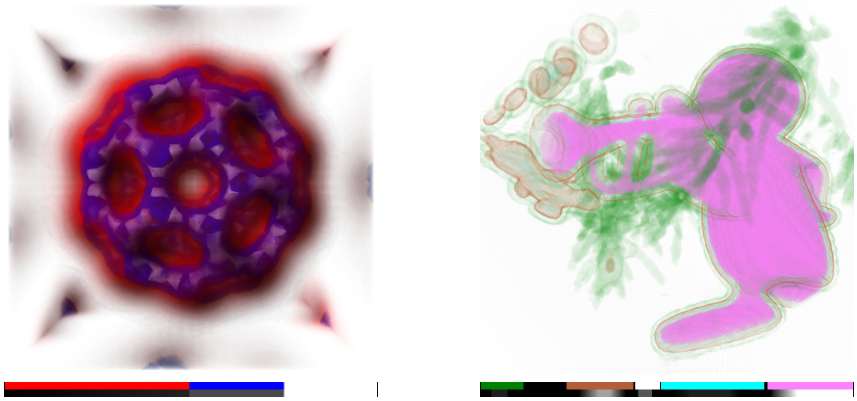


Figure 9.6: Bucky Ball with per-vertex lighting of original data set and part of the Christmas Tree data set [46], both with quadratic polynomial approximation of chromaticity and accurate 16 bit α.

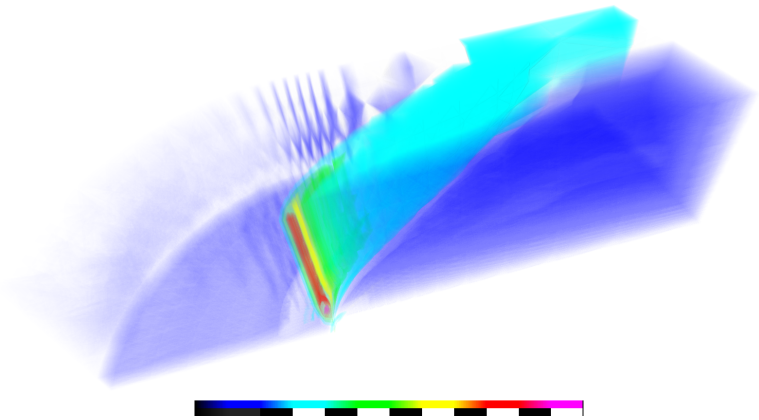


Figure 9.7: Blunt Fin dataset using quadratic polynomial approximation of chromaticity and 16 bit α. Due to the high reconstruction quality of the pre-integration table, the undersampling within the original data set can easily be seen.

Chapter 10
Ground Fog Rendering

Albert Einstein: *As far as the laws of mathematics*
refer to reality, they are not certain;
and as far as they are certain,
they do not refer to reality.

In contrast to Chapters 8 and 9, where the pre-integration technique was developed to enhance the quality of unstructured volume rendering, in this chapter we try to improve the performance by posing several restrictions on the optical model. The performance of the resulting algorithm is demonstrated by rendering ground fog in real time.

Actual implementations of the PT algorithm achieve a peak performance of 250,000 [121] to 600,000 [35] tetrahedra per second including times for sorting. Due to the growing complexity and amount of the data sets frame rates of less than one frame per second are still quite common for typical unstructured data sets.

Recently, hardware-accelerated methods have been proposed to speed up the PT algorithm, but with actual graphics hardware still no more than approximately 480,000 [109] respectively 490,000 [124] tetrahedra are possible. In fact, the hardware-accelerated methods are slower than a well tuned software implementation. This observation may be astonishing in the first place, but with an in depth analysis of the bottlenecks this becomes clear. There are four limiting operations which determine the performance of the PT algorithm: visibility sorting, tetrahedral projection, transfer of vertex data, and finally rasterization. Using a fast CPU like the Pentium 4 3.0 GHz, the ordering and the projection of the tetrahedra is performed faster than the GPU can be fed over the AGP bus. With the increasing speed of the GPUs the vertex processing performance is almost balanced with the performance of the CPU, but rasterization still is a limiting factor. In conclusion, performing the cell-projection on the GPU will only slow down the entire pipeline, since the graphics processor is already near its limit. There also exist hardware concepts to overcome the speed limitations, but it is uncertain when these concepts will find its way into graphics accelerators [47].

Since recent efforts to significantly speed up the PT algorithm have not led to satisfactory results, we pursue a different strategy: First we evaluate the theoretical speed limit on the number of polyhedra that can be rendered on actual

graphics hardware. Based on these results we propose a reasonable modification of the optical model to approach the theoretical limit.

10.1 Theoretical Performance

In principle, all the faces of an unstructured data set have to be treated to reconstruct the ray integral exactly. For the case of hexahedral cells, this results in 6 faces with 4 vertices each. Assuming that the volumetric grid can be rendered with triangle stripping, 8 vertices have to be passed down the graphics pipeline per hexahedron. Actual graphics adaptors like the NVIDIA GeForce3 reach a peak performance of about 12 million vertices per second using triangle strips (in practical experience). Thus, the maximum theoretical performance of a GeForce3 is 1.5 million hexahedra per second.

In order to verify the theoretical result, we first applied maximum intensity projection (MIP) [38]. The advantage of MIP is that a volumetric grid can be visualized just by rendering all the faces of the cells in an unsorted order. Without great loss of accuracy the scalar values can be assumed to vary linearly inside each hexahedron. Then the maximum projected scalar value of each ray segment is either the value at the intersection point on the front or on the back face. Using this approach we achieved a performance of 643,000 hexahedra or 5.1 million triangles per second. Assuming that a hexahedron needs to be decomposed into at least 5 tetrahedra to be rendered with the PT algorithm the experimental result of 643,000 hexahedra per second corresponds to 3.2 million tetrahedra per second. This is still far away from the theoretical maximum, but it is almost a magnitude faster than the best known PT implementation.

10.2 Practical Performance Analysis

The performance for such a simple optical model like MIP is already considerably lower than the theoretical limit. This is mainly due to the large rasterization overhead. Hence, it is no surprise that the performance is even worse in the case of the standard volume density optical model [119]. This is due to the requirement of visibility sorting. Conceptually, the tetrahedra must be read, written, and read back from main memory for sorting (compare Wittenbrink et. al [121]). With increasing rendering speed of the graphics accelerator the memory bandwidth consumed by visibility sorting and data transfer over the AGP bus becomes the limiting factor. This behavior starts at approximately 1.5 million tetrahedra per second on actual PC hardware. Since the total performance is currently only around 600,000 tetrahedra per second the main limiting factor is still the graphics

accelerator. We suspect that a significant performance bump beyond the mentioned 1.5 million tetrahedra per second limit is possible only with a structural paradigm shift of graphics accelerators or special purpose hardware.

Because of the limiting behavior of visibility sorting, we devise an efficient algorithm for an emissive optical model[68] which does not require sorting. In our opinion this optical model can be considered to be a good tradeoff between speed and quality. The emissive optical model neglects absorption so that the ray integral is simply the sum of all emissions along each ray. As a welcome side effect sorting is not required, since the blend function is commutative. In comparison to the standard optical model the emissive model gives less visual clues but as we will see the implementation is extremely simple so that it can serve as a fast preview and prototyping option.

Recently, Mech[70] proposed a method to render bounded layered fog using an emissive optical model. The bounded fog is defined within a triangular surface mesh which allows for easy hardware-accelerated computation of the ray integral (see Section 5.4). While this approach is simple yet very fast, it assumes a constant fog density and requires a 12 bit visual to eliminate Mach bands. In the following we extend this algorithm to project arbitrary cell types, such as tetrahedra, hexahedra, or prisms, without the restriction to a 12 bit frame buffer and with linearly interpolated densities within each cell.

10.3 Projected Convex Polyhedra Algorithm (PCP)

Our so-called Projected Convex Polyhedra (PCP) algorithm requires three passes per cell. In the first two passes the length of the ray segments is calculated in the alpha channel of the frame buffer. For this purpose, the distance d to the near plane is computed for each vertex of the cell. Let d_{max} denote the maximum distance, let d_{min} denote the minimum distance, and let $\Delta d = d_{max} - d_{min}$ be the difference of both (see also Figure 10.1). Then the back faces of a cell are rendered into the alpha channel of the frame buffer with the alpha component of each vertex set to $\alpha = (d - d_{min})/\Delta d$. In the same fashion, the front faces of the cell are rendered into the alpha channel with subtractive blending enabled. As a result, the normalized ray segment lengths are now available in the alpha channel of the frame buffer. In the last pass all faces of the cell are rendered into the color channel of the frame buffer. Let $\kappa(S)$ denote the transfer function of the emissive optical model depending on the scalar value S. Then the color I of each vertex is set to $I = \frac{1}{2}\kappa(S)\Delta d$. The following blend function is applied as described in OpenGL notation: `glBlendFunc(GL_DST_ALPHA,GL_ONE)`. This effectively multiplies the average emission along each ray segment with the segment length already stored in the alpha channel.

In contrast to Mech's method we do not require a 12 bit visual, since we use normalized ray segment lengths for each cell. A solution to suppress the Mach bands, if we would apply Mech's method, would be to use the floating point render target of actual PC graphics accelerators such as the ATI Radeon 9700 or NVIDIA NV30. However, since the algorithm is mainly rasterization bound the increased bandwidth for the floating point render target would significantly slow down rendering. Additionally, Mech's algorithm is not as flexible as ours. Using our method, almost any desired volumetric object or effect can be constructed from convex polyhedra, such as tetrahedra, prisms, and hexahedra in a very compact way.

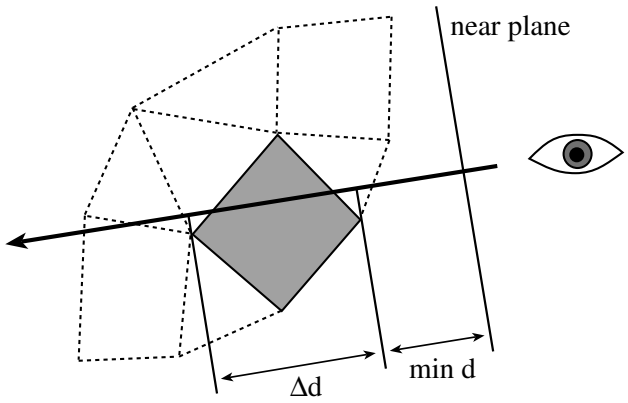


Figure 10.1: Projection of polyhedral cells.

In principle, all types of cells used for FEM such as tetrahedra, hexahedra, prisms, pyramids etc. are compatible with our approach. For the common case of projected hexahedra Schussman et al. [96] report about 80,000 hexahedra per second. We achieve about 212,000 hexahedra per second, which is a performance increase of 265%. Compared to the 643,000 hexahedra per second of the MIP method, the performance drop is mainly due to the increased number of rendering passes.

In Figure 10.2 and 10.3 example data sets are shown that have been visualized with the PCP algorithm. The corresponding timings are given in Table 10.4. To speed up projection hexahedra with zero emission were discarded.

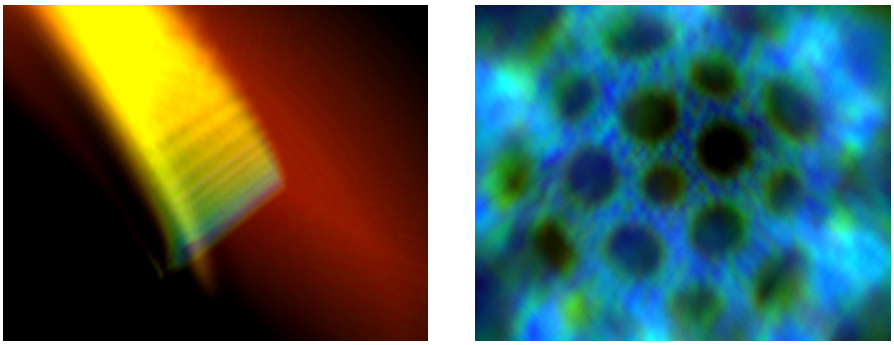


Figure 10.2: Blunt Fin and Bucky Ball data set rendered with the PCP method.

10.4 Application of the PCP Method to Ground Fog Rendering

Ren Höek: *Back off man!!!...*
Don't make me use this...
One step closer, I'm warning ya...
Don't make me use it!
Now you've done it!
You... forced me to use it!!!...

Eventually, we come to the first application example of the proposed unstructured volume rendering methods in the area of natural gaseous phenomena. Besides the application area of scientific volume visualization as demonstrated in Figure 10.2 the performance and flexibility of the proposed cell projection algorithm in Section 10.3 paves the way for other fields of application. As an example, we demonstrate the real time display of ground fog. In principle, all effects related to light emitting gas can be modeled. In particular, the display of ground fog in terrain rendering scenarios benefits from our algorithm, as shown in the following.

In a terrain rendering scenario the landscape is commonly given as a height field (see Chapter 3). Here, the basic idea to display ground fog is to use a second height field (the ground fog map) which defines the height of the fog layer above the ground. Each triangle of the surface mesh is treated as a base triangle onto which a vertically aligned prism is stacked. The height of the prisms, that is the heights of the three vertical edges of each prism, are derived from the ground fog map (see Figure 10.5).

At the top left of Figure 10.6 an example height field of Yukon Territory, Canada, is depicted. The shown ground fog has been generated with 2D Perlin noise [82]. In order to reduce the number of displayed triangles and stacked

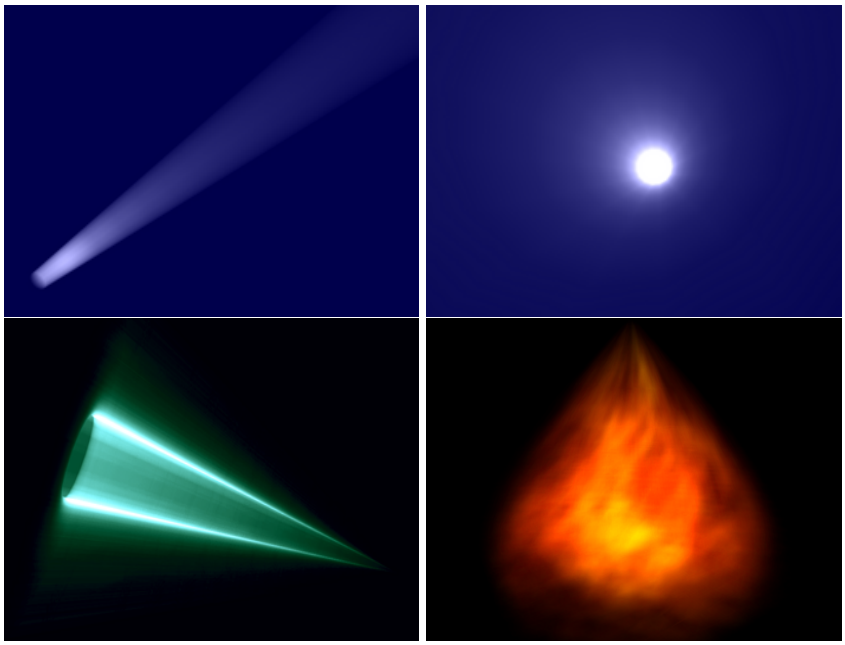


Figure 10.3: Synthetic data sets: A search light with quadratic intensity attenuation, a laser cone, and a campfire generated with 3D Perlin noise.

prisms, we used a C-LOD approach (see Chapter 3), that is our C-LOD implementation of [89]. The triangulation algorithm of this C-LOD implementation is driven by a subdivision criterion which depends on viewing distance and local curvature. In case of ground fog rendering the local curvature of both the height field and the ground fog map has to be considered. Since the projection of a prism takes more time than rendering the base triangle, the local curvatures are not considered equally important. In our approach the maximum of the weighted local curvatures is taken as subdivision criterion. The C-LOD algorithm also implements geomorphing so that the popping effect is suppressed efficiently. This allows the viewer to fly through the ground fog without experiencing any temporal aliasing artifacts.

Figure 10.4: Timings for hexahedral projection.

Data set	dimension	#hexahedra	frames per sec.
BluntFin	32×32×40	37,479	8.5
Bucky Ball	32×32×32	29,791	15.9
Search Light	16×4×32	1,395	115.8
Camp Fire	16×16×16	3375	51.3

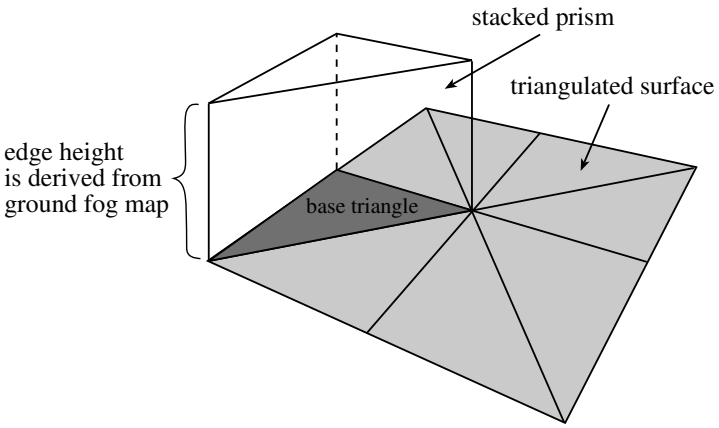


Figure 10.5: Stacking prisms onto a triangulated surface.

We achieved an average frame rate of approximately 25 Hertz for a window size of 512×384.

Inside the fog, we have to take care of prisms that intersect the near clipping plane. In such a case the ray segment lengths are partially invalid, since the corresponding back faces are not rendered completely. To circumvent this problem, we also render the intersection of each prism with the near clipping plane with $\alpha = (d_{near} - d_{min})/\Delta d$ after the second pass. The same strategy is necessary for the third pass.

The ground fog in the valley as shown at the top right of Figure 10.6 is displayed with maximum intensity projection. The corresponding height field has been painted by hand with a standard image manipulation application. Since the MIP method requires only one pass in comparison to the three passes of the PCP algorithm the rasterization bottleneck is reduced significantly. This leads to more than twice the frame rate (> 50 Hz) as in the previous example.

Despite the seemingly unsuitable optical model we have found a reasonable setup for the MIP method: The fog's optical density is set to zero at the bottom of the prisms. At the top of the prisms the density correlates to the height of the fog layer. Despite the fact that this setting does not reproduce the fog physically correct it is well suited for the real time display of foggy areas in computer games where fog can be used as a game play relevant element.

Another application area of the described ground fog renderer is the display of the Aurea Borealis (polar light). We simply set up a height field that corresponds to the penetration depth of the particles into the earth's ionosphere (see Figure 10.6).

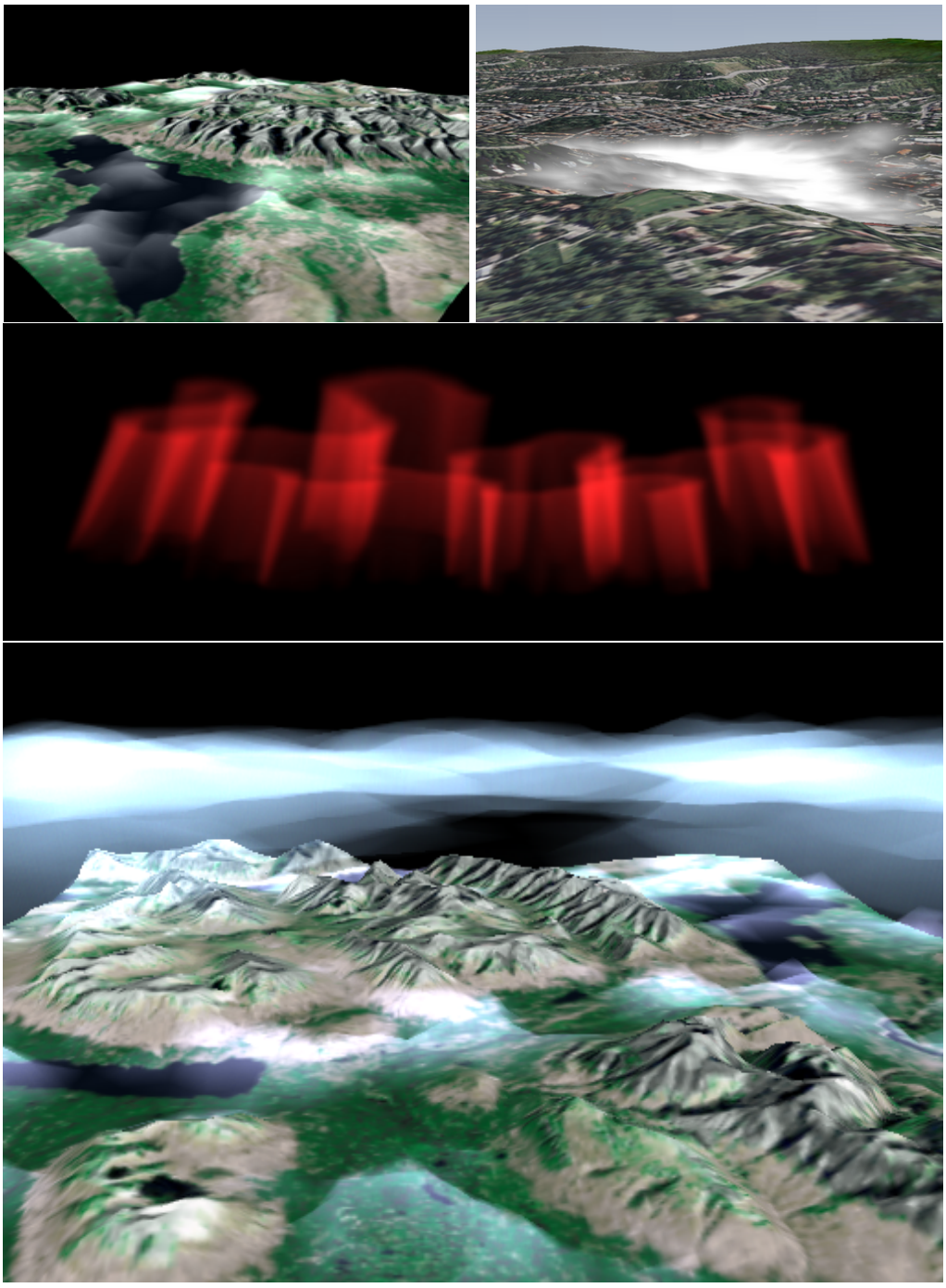


Figure 10.6: Ground fog generated with 2D Perlin noise. **Top left**: Emissive optical model. **Top right**: Maximum Intensity Projection (MIP). **Centre:** Aurea Borealis (polar light). **Bottom**: Ground fog in combination with a cloud layer, which is defined by two height fields.

Chapter 11
Cloud Rendering

The PCP algorithm employs an emissive optical model or MIP, which results in unprecedented performance but is not realistic in a physical sense. Due to its flexibility, it is also suited for the display of volumetric effects in interactive entertainment. We have demonstrated this by rendering fire and ground fog in real time. In this chapter we apply the pre-integration methods developed in Chapters 8 and 9 to achieve a higher quality. In particular, we demonstrate real time cloud rendering. In order to achieve real time performance we also develop a view-dependent mesh simplification scheme which reduces the number of rendered tetrahedra for large cloud data sets. This is the analogue procedure as using the C-LOD algorithm for the display of ground fog.

11.1 View-Dependent Rendering

In general, the strategy to simplify a mesh in a view-dependent fashion is suited well for the real time display of large scenes [126]. This has been exemplified by the C-LOD algorithms in the area of terrain rendering as shown in Chapter 3. The C-LOD technique achieves high frame rates by generating an approximate view-dependent triangulation of the terrain. In order to minimize the total screen space error of the approximation, small distant details are represented with fewer triangles than those which are nearby.

Despite the widespread use and the maturity of the C-LOD technique it has not yet been applied to the more general case of volume rendering: A multi-resolution analysis for the display of polygonal meshes has been introduced by Rossignac et al. [91], and has been the subject of intense studies later on (see Xia et al. [126] as a starting point). General multi-resolution analysis of volumetric meshes has been given by Eck et al. [24] and more recently by Cignoni et al. [11]. Variants for the hierarchical visualization of regular volume data have been discussed by Laur et al. [56], Zhou et al. [129], and Schussman et al. [96]. And finally a view-dependent simplification method for irregular grids has been proposed by Meredith et al. [73]. But the efficient view-dependent simplification of regular volume data is still an active research field.

11.2 C-LOD Rendering

In the following we present a general purpose volume rendering algorithm which is based on the continuous level of detail idea. It maintains an octree to construct a view-dependent representation of regular volume data. After decomposing each leave node of the octree into tetrahedra these can be rendered efficiently using the projected tetrahedra algorithm of Shirley and Tuchman [101].

A common property of view-dependent algorithms is the occurrence of the so-called popping artifacts: Small distant details will suddenly pop up when approaching nearby. In the case of a C-LOD terrain renderer the total screen space error of the approximation can be pushed easily below the one pixel boundary, so that the popping effect becomes invisible. In the volumetric case, however, this approach is infeasible. As a solution to this problem, the mesh hierarchy has to be interpolated smoothly. In consideration of this fact, we present a fast mesh interpolation method, which we refer to as volumetric morphing throughout this chapter.

11.3 Generating Continuous Levels of Detail

In this section we describe how to adapt the C-LOD technique previously known from terrain rendering [59, 19, 89] to the volumetric case.

11.3.1 Hierarchical Volume Representation

Given a three-dimensional scalar field, which is defined by an array with $2^n + 1$ ($n > 0$) grid points in each dimension, a hierarchical volumetric mesh is constructed by building an octree in a bottom-up fashion. Grids with a size other than $2^n + 1$ have to be padded or resampled. Each leave node of the octree is decomposed into five tetrahedra. Since there exist two topologically different decompositions, adjacent nodes of the same level of detail have to be decomposed in an alternating fashion to ensure a conforming mesh. In Figure 11.1 the orientation of the tetrahedra is depicted for a coarse example hierarchy with a total of three different levels of detail.

11.3.2 View-Dependent Mesh Simplification

The key idea of a volumetric C-LOD algorithm can be described as follows: In order to perform a view-dependent simplification the octree has to be updated for each frame. During a top-down traversal of the octree our approach calculates an

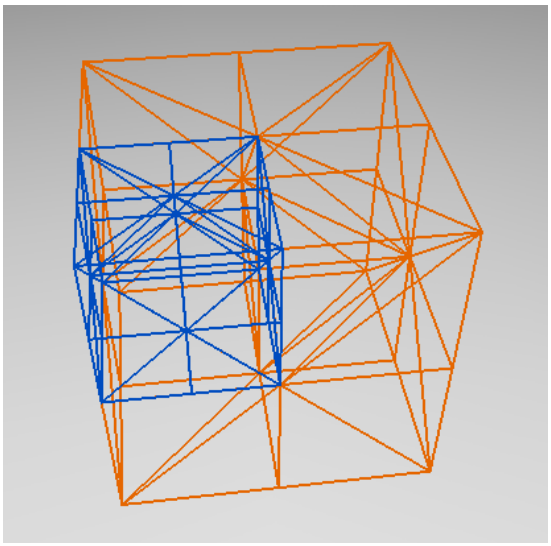


Figure 11.1: Hierarchical volume representation using an octree: The example hierarchy consists of the root node with 8 children (bright/orange), one of which has been refined into another 8 children (dark/blue). Each leave node of the octree is decomposed into five tetrahedra in an alternating fashion.

upper limit on the local screen space error of each node. If the local error exceeds a predefined threshold the corresponding node is split into eight children.

The error metric used to estimate the local screen space error is designed to meet the following criteria: A node should be refined if the local simplification error is large. Also, small distant nodes should be refined less likely than those which are nearby. Let s be the edge length of each node, let d denote the euclidean distance of the eye to the center of the node, and let Δ be the local simplification error of the node in object space. With the previous definitions, we introduce the absolute error E, the base error b, and the normalized error $e = E/b$ as follows:

$$e = \frac{sC\max(c\Delta, 1)}{d} \tag{11.1}$$

If the normalized error e is greater than one (meaning that the actual error is greater than the base error b), the node is refined, else the refinement of the octree is stopped. The base error is set indirectly by choosing appropriate values for the two constants c and C. Typically, C is chosen to be constant, so that by tuning c in the range from $[1, \infty[$ the resolution of the mesh can be adapted conveniently. Higher values of c result in a finer mesh, whereas the constant C defines the minimum possible resolution.

In a preprocessing step the local error Δ is computed. It is defined to be the average of the scalar deviations Δ_i at the center of the node and the midpoints of the edges and faces. The scalar deviations Δ_i are equal to the difference of the scalar value of each vertex and the interpolated scalar value derived from the next coarser level of detail. For instance, the deviation of the midpoint of an edge is equal to the absolute scalar difference of that vertex and the average of the two adjacent corner vertices (also compare Figure 11.2 where $\Delta_{midleft} = |\frac{1}{2}(S_{topleft} + S_{bottomleft}) - S_{midleft}|$).

11.3.3 Building a Conforming Mesh

For adjacent nodes, which do not belong to the same level of detail (depicted by the orange and blue colors in Figure 11.1), the interpolated scalar values at a T-vertex of the boundary face do not match. One solution to ensure a conforming mesh is to insert irregular tetrahedra into the coarser node. This technique is known as the red-green or regular-irregular refinement method [2, 34]. But if we want to morph between two of the large number of irregular configurations, the situation is getting inscrutable complex. Furthermore, adjacent nodes must not differ by more than one level for this method to work. In order to circumvent these problems, we employ a different approach: Rather than inserting irregular tetrahedra into the coarser node, we manipulate the scalar values of the refined node. To build a conforming mesh the scalar value at a T-vertex is simply substituted by the interpolated value from the coarser mesh of the adjacent neighbor node. A detailed example is given in Section 11.4.

11.3.4 Hierarchical Error Propagation

Since we use a top-down simplification approach, at each node only the local simplification error is known. However, in order to minimize the total screen space error of the generated mesh, we also need to know the local simplification error of all children in advance. This can be accomplished by propagating the local error from the children up to the parents of the octree. In principle, the error propagation has to ensure that a node is refined, if at least one child already fulfills the refinement condition. In mathematical terms this can be written as:

$$e_{child} > 1 \rightarrow e > 1 \quad \text{or} \quad e > e_{child} \tag{11.2}$$

Substituting Equation 11.1 into Equation 11.2 yields

$$\Delta > K\Delta_{child} \quad \text{with} \quad K = \frac{d}{2d_{child}}. \tag{11.3}$$

Now we determine an upper bound for K. Since we introduced a minimum accuracy C which always guarantees refinement for $d < sC$ we just have to consider the case $d \geq sC$. On the one hand, the minimum possible value of K is $\frac{1}{2}$ for an infinite distant viewer. On the other hand, the maximum possible value of K occurs for the minimum distance $d = sC$. Then the minimum distance to the center of one of its children is $d_{child} = sC - \frac{1}{4}\sqrt{3}s$. Resubstituting these distances into Equation 11.3 yields the following upper bound for K:

$$K_{max} = \frac{C}{2C - \frac{1}{2}\sqrt{3}} \quad (C > \sqrt{3}) \tag{11.4}$$

As a consequence, Formula 11.5 can be used to propagate the local error Δ from all eight children up to the parent nodes. Starting with the leave nodes, all nodes which belong to the same level of detail are processed in a row. For each node the final propagated Δ-values are stored at the center vertex of each node using a linear mapping with 16 bits of accuracy.

$$\Delta := \max(\Delta, K_{max} \cdot \Delta_{child}) \tag{11.5}$$

In summary, the update of the view-dependent hierarchy is performed by refining the octree, if and only if the normalized error defined in Equation 11.1 is greater than one. The simplicity of this approach is the basis for the real time performance of our algorithm. Another important advantage is that volumetric morphing can be implemented very efficiently as shown in Section 11.4.

11.4 Volumetric Morphing

Arthur C. Clarke: *Any smoothly functioning technology will have the appearance of magic.*

In this section we describe a new fast method to morph the view-dependent hierarchy. Volumetric morphing is mandatory, because otherwise the transition from one level of detail to another could be observed easily.

For each frame, first the hierarchy is updated using the view-dependent approach described in Section 11.3. During the update the error metric e is mapped to the range $[0,1]$ according to Equation 11.6 and stored at the center vertex of each node with 8 bits of accuracy.

$$e' = \min(\max(e-1,0),1) \tag{11.6}$$

In a second octree traversal, the normalized error metric e' is interpreted in the following way: A value of zero ($e \leq 1$) means that the corresponding node has

not yet been refined, thus it can be decomposed into 5 tetrahedra and rendered as described in Section 11.5. A value greater than zero and less than one ($e \in (1,2)$) means that the node has been refined but still none of its children. A value of one ($e \geq 2$) means that the node and at least one of its children have been refined. As a consequence, the time between the two subsequent refinement events for $e' = 0$ and $e' = 1$ can be used to blend the scalar values of the corresponding node as smooth as possible. Thus, the parameter e' just serves as an interpolation factor to morph recursively between the actual node and its children.

In contrast to a fixed blending time interval [42], the speed of the interpolation is coupled to the error metric. This is a much better strategy for volumetric morphing, since distant details can be morphed much slower than those which are nearby. In practice, we have found that the maximum instead of the average of the deviations Δ_i suppresses the popping effect more reliably. This is due to the fact that the subjective observability of the interpolation is determined by the maximum and not by the average change of all vertices.

In the context of the described interpolation scheme a conforming mesh can be guaranteed simply by using the minimum interpolation factor of all adjacent nodes which share the interpolated vertex. If one of the adjacent nodes has not been refined, the corresponding interpolation factor is assumed to be zero.

In the following we illustrate the described interpolation scheme using a two-dimensional example, which is depicted in Figure 11.2. In general, only the scalar values of the non-corner vertices of a node have to be interpolated using the normalized error metric e' as the interpolation factor. In the two-dimensional case, the non-corner vertices of a node are the midpoints of the four edges (black dots) and the center vertex (white dot). For each of those vertices the interpolation is performed between the average scalar value of the two adjacent corner vertices (small black crosses) and the actual scalar value of the vertex. For the midpoint of the left edge of the grey node in Figure 11.2, for example, the interpolated scalar value $S'_{midleft}$ is calculated as follows:

$$w = \min(e', e'_n) \tag{11.7}$$

$$S'_{midleft} = w\frac{1}{2}(S_{topleft} + S_{bottomleft}) + (1-w)S_{midleft} \tag{11.8}$$

In the three-dimensional case, the non-corner vertices of a node are the centroid, the midpoints of the six faces, and the midpoints of the eight edges. The scalar values of these vertices are interpolated in analogy to the two-dimensional example. The midpoints of the edges are shared among the actual node and a maximum of three adjacent nodes of the same level of detail. Thus, the minimum interpolation factor of these vertices has to be calculated from the interpolation factors of the actual and the three adjacent nodes. The midpoints of the faces are shared among two nodes. Here, additional care must be given to the calculation of the

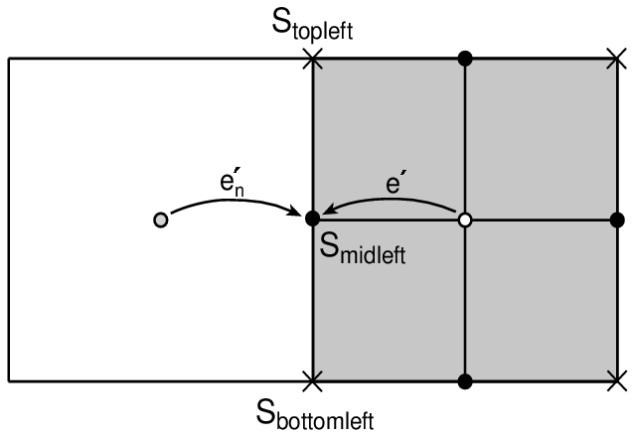


Figure 11.2: Two-dimensional morphing example.

average scalar value of the corner vertices, since adjacent nodes are decomposed in an alternating fashion. The average scalar value is therefore computed from the appropriate two corner vertices of the tetrahedral decomposition of the adjacent neighbor node. The centroid of a node is located inside the center tetrahedron of the decomposition. In this case the average scalar value is computed from the four vertices of the center tetrahedron. Again, special care must be given to the calculation of the average scalar value, since the orientation of the center tetrahedron alters.

11.5 Cell Projection

Now that we have performed a view-dependent simplification of a regular volume, the generated tetrahedra have to be composed in a back to front fashion. We apply the cell-projection technique, that is the PT algorithm of Shirley and Tuchman [101, 106, 116, 120]. The original PT algorithm only supports linear transfer functions which are not appropriate for the display of gaseous phenomena as demonstrated in Section 11.6. Therefore we also apply the pre-integration method introduced in Chapter 8 (compare also [90, 69, 26]) which allows the use of arbitrary transfer functions by storing the ray integral in a three-dimensional pre-integration table. Visibility sorting [118, 13] is performed by reordering the traversal of the octree in a back to front fashion. For this purpose the children of each node are traversed back to front. This ensures that at each level of detail the nodes are depth sorted, which is equivalent to a total ordering of the octree.

In order to speed up the PT algorithm we discard transparent tetrahedra by applying the so-called *Zero Opacity Test* (ZOT). While this test is obvious for

linear transfer functions, it is not as obvious for arbitrary transfer functions. Fortunately, the three-dimensional pre-integration table contains all necessary information to apply this test. First, the minimum and maximum scalar values (denoted by S_{min} and S_{max}) of the tested tetrahedra are computed. If the entry at position $(\lfloor (n-1)S_{min} \rfloor, \lceil (n-1)S_{max} \rceil, m-1)$ of the pre-integration table (with size $n \times n \times m$) is zero, then we can discard the tested tetrahedra. By applying the ZOT to each visited node of the octree, we can discard all transparent tetrahedra with virtually no computational overhead.

Another common way to speed up rendering is view frustum culling. During the rendering traversal of the octree each node is tested against intersection with the view frustum. If a node does not overlap with the view frustum, it is invisible and can be discarded.

Since we want to allow the viewer to navigate freely inside the volume, we face the following problem: If a tetrahedron intersects the near clipping plane, the clipped two-dimensional projection is not identical with the clipped volume of the tetrahedron. In order to display the tetrahedron correctly, it has to be be clipped in a truly volumetric fashion. We distinguish two different cases: Either the tetrahedron is cut into one tetrahedron and one prism or it is cut into two prisms. Therefore, the visible part of the clipped tetrahedron is either a tetrahedron or a prism. In the latter case the total number of rendered tetrahedra is increased, since the prism has to be decomposed into three tetrahedra. However, in comparison to the total number of rendered tetrahedra, the number of clipped tetrahedra is fairly low. Therefore the number of additionally generated tetrahedra is uncritical.

11.6 Non-Photorealistic Cloud Rendering

In the previous sections we have described a general purpose volume rendering algorithm which is based on a view-dependent simplification. In this section we demonstrate the abilities of this approach by rendering volumetric clouds.

In general, we can think of a cloud as a three-dimensional scalar function $f = f(x,y,z)$. The scalar values correspond to the optical density of the medium. Due to the complex anisotropic light scattering [3, 31, 23, 67, 105, 78, 43] inside a cloud the photorealistic display is a time consuming task. Impostors [95, 97] are currently the dominating technique here [16, 25, 37] (see also Chapters 5 and 6).

However, if we restrict ourselves to isotropic light scattering the cloud intensities can be precomputed and we can apply the described view-dependent simplification algorithm. As a result, the clouds are modeled by two scalar fields, the scalar density $f(x,y,z)$ and the scalar isotropic light intensity $\gamma(x,y,z)$. The mesh simplification is driven by the maximum deviation of both scalar fields. This approach has the following advantages: Since we use a truly volumetric rep-

resentation there are no restrictions with respect to cloud shape and appearance. As opposed to the impostor method, the clouds are displayed without temporal aliasing or perspective artifacts, even for view points inside the clouds. This is guaranteed by the application of volumetric morphing and clipping.

11.6.1 Modified PT Algorithm

The volume density optical model [119] used for pre-integrated volume rendering presumes the transfer functions κ (the chromaticity vector) and ρ (the scalar optical density) to depend both on the scalar density function $f(x,y,z)$. But, since we want the optical density to depend on the density function $f(x,y,z)$ and the chromaticity vector to depend on the precomputed light intensities $\gamma(x,y,z)$, we circumvent this restriction of the optical model by slightly modifying the PT algorithm. For this purpose, we assume that $\rho(f) = f$. Then we apply the pre-integration to the chromaticity vector $\kappa = \kappa(\gamma(x,y,z))$ and the maximum optical density $\rho_{max} = f_{max}$. To introduce the dependency on $f(x,y,z)$ we modulate the effective length l of each tetrahedral ray segment by the scalar optical density $\rho = f(x,y,z)$ according to the following equation:

$$l' = l\frac{f(x,y,z)}{f_{max}} \tag{11.9}$$

11.6.2 Non-Photorealistic Lighting

The previous approach requires an isotropic light scattering simulation [67, 78, 43] to calculate the light intensity function $\gamma(x,y,z)$. Instead of determining physical simulation parameters we propose a non-photorealistic approach which achieves the desired look and feel of the clouds by a direct manipulation of the transfer functions.

For this purpose, the chromaticity vector $\kappa = \kappa(f(x,y,z))$ is defined to be an inverse color ramp and the optical density $\rho = \rho(f(x,y,z))$ is defined to be a linear function except for very small densities where it is set to zero. This allows to speed up rasterization by discarding nearly transparent areas with the ZOT. The light intensities γ are calculated by standard ambient and diffuse lighting and are used to modulate the effective ray segment length as described before. With respect to the direction of the diffuse light this leads to high opacities at the front and to low opacities at the back of the clouds. As a consequence, the dark inside of each cloud shines through the translucent back, but at the front bright colors still dominate the appearance of the clouds. This approach effectively mimics the natural look and feel of clouds without requiring a physical lighting simulation.

11.7 Performance Measurements

Using the described non-photorealistic cloud and ground fog rendering algorithms, Figures 11.3 and 12.4 show the city center of Stuttgart with some cumulus clouds and ground fog in the valleys. The scene was rendered in real time with approximately 26 frames per second on a PC equipped with a 1.2 GHz AMD Athlon and an NVIDIA GeForce3 graphics adaptor. About 25% of the total rendering time was spent on terrain rendering [89], 20% was spent for the display of the ground fog and the remainder of 55% for the display of the clouds. The latter were generated with 3D Perlin noise [82, 22]. The applied transfer functions κ and ρ are depicted on the left side of Figure 11.3. The ease of changing the appearance of the clouds by choosing different transfer functions is illustrated in Figure 11.4 showing the Big Island of Hawaii during daytime and sunset.

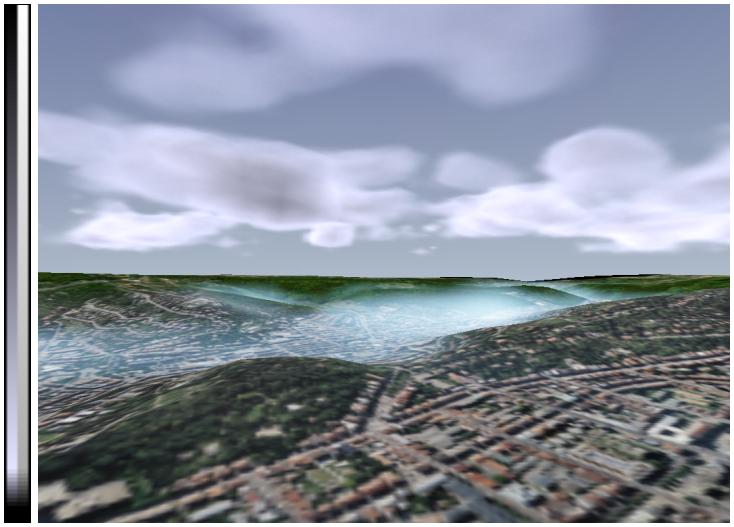


Figure 11.3: The city center of Stuttgart with clouds and ground fog in the valleys. The applied transfer functions κ and ρ are depicted on the left.

The size of the height field and the ground fog map is 2049×2049, whereas the cumulus clouds are represented by an 8 bit density field with a base size of 513×513 and a height of 65 grid points. For the density field one byte is consumed per grid point plus 16 bits for the deviations Δ and one byte for the interpolation parameter e' summing up to a total of 48 MB in our example. The size of the pre-integrated 3D texture is $64 \times 64 \times 128$ which corresponds to 2 MB of graphics memory. Since only the 3D texture has to be kept in graphics memory, the maximum cloud size is limited by main memory only. For the display of the clouds the number of rendered tetrahedra was reduced from a total of 83 million to

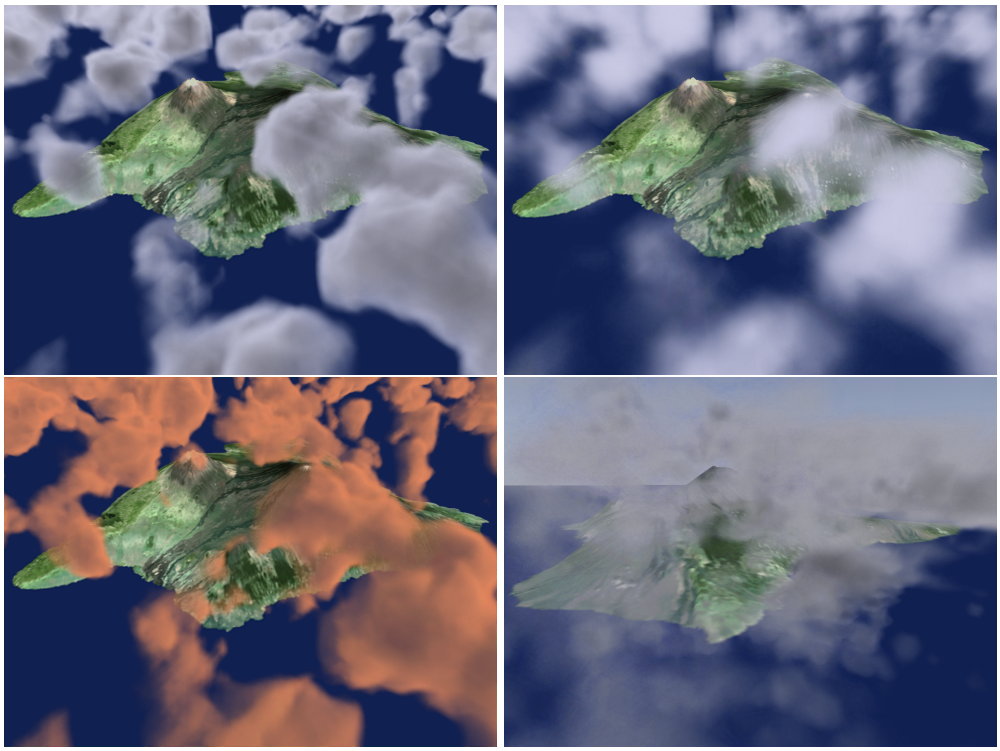


Figure 11.4: The impact of different transfer functions on the appearance of the Big Island of Hawaii. **From top left to bottom right:** Perlin noise clouds with pre-integration and lighting, linear transfer function without lighting, sunset-like transfer function, and simulated bad weather.

less than 10 thousand tetrahedra on the average. This corresponds to a reduction of four orders of magnitude.

An analysis of the experimental results reveals two bottlenecks. The main bottleneck is the projection of the tetrahedra. This is due to the fact that for each single node of the octree 5 tetrahedra have to be decomposed into an average number of 17.5 triangles. If the view point is entirely inside a cloud, the algorithm is mostly fill-rate bound and the performance drops to approximately 15 frames per second for a window size of 512×384 pixels.

11.8 Discussion

In comparison to the impostor technique, our approach offers the following advantages: Most important, a flight through the clouds does not introduce temporal aliasing or perspective artifacts, since we use volumetric morphing and clipping.

Furthermore, our general purpose volume rendering algorithm is able to render arbitrary weather conditions including overcast sky and thunder storm clouds (see Figure 12.2). Besides the shown 3D Perlin noise example more sophisticated cloud simulation algorithms [77, 75, 29] are compatible with our approach, which makes the algorithm well suited for the purpose of weather visualization.

In Figure 11.5 we have tried to match the virtual view of the city center of Stuttgart with the actual real view as seen from the vista point called "Birkenkopf". Of course, one can clearly see the difference between the real and the virtual image, since it is very difficult to reproduce the vegetation on the ground. It is even more difficult to model the captured real weather situation. This would require a deep understanding of weather simulation which is not the aim of this thesis. Nevertheless, in principal, the most important natural volumetric phenomena such as clouds and ground fog can be displayed at real time. Using more sophisticated simulation methods and real input data would lead to greatly improved realism. This has been shown by Thomas Gerstner et al. [32] who displayed real weather radar data by means of an hierarchical cell-projection approach. Unfortunately, weather radar data captures only the precipitation distribution and not the detailed shape of the rain clouds.

Figure 11.5: Real and virtual panorama of Stuttgart.

Chapter 12

Summary

In this thesis our aim was to develop algorithms for the real time display of natural gaseous phenomena, that is particularly ground fog and clouds. We analyzed the existing methods in this research area and came to the conclusion that truly volumetric methods were lacking. We developed unstructured volume rendering techniques that were mainly aimed at scientific volume rendering in the first place. But we have demonstrated that unstructured volume rendering methods are also suitable for reaching our goal of displaying natural phenomena at real time.

To be more precise, we developed two real time volume rendering methods in this thesis. The first one is the **PCP method**, and the second one is **pre-integrated cell-projection**.

Right now, the first one, the PCP technique, can be used in interactive entertainment to model volumetric effects with greater realism. Typical application areas are the display of fire, search lights and the like (see Figure 10.3). The use of a non-photorealistic rendering model somewhat limits the area of application, but in many cases the real time performance outweighs this restriction. In interactive entertainment the layered fog technique [57, 39] is used commonly (i.e. in the DX8 game AquaNox [87]), but has the disadvantage that the vertical fog boundaries are fixed. With the ground fog rendering algorithm described in Chapter 10 we overcome this restriction by explicitly defining the height of the fog layer. In order to reduce the geometric complexity we not only apply the C-LOD approach to the terrain but also to the ground fog layer. The resulting view-dependent representation allows to minimize the otherwise inherently large volume rendering overhead. This is the first time an approach has been presented which does not restrict the ground fog model in a geometrical sense. An example is shown in Figure 12.1.

The second method for visualizing volumetric effects at real time is pre-integrated cell-projection. It allows to reconstruct the ray integral with per-pixel exactness. Previous unstructured volume rendering approaches were either strongly limited in accuracy or in rendering speed. Pre-integrated cell-projection achieves both high quality and high performance. We combined this approach with a three-dimensional version of the well-known continuous level of detail technique, which approximates a three-dimensional scalar field in a view-

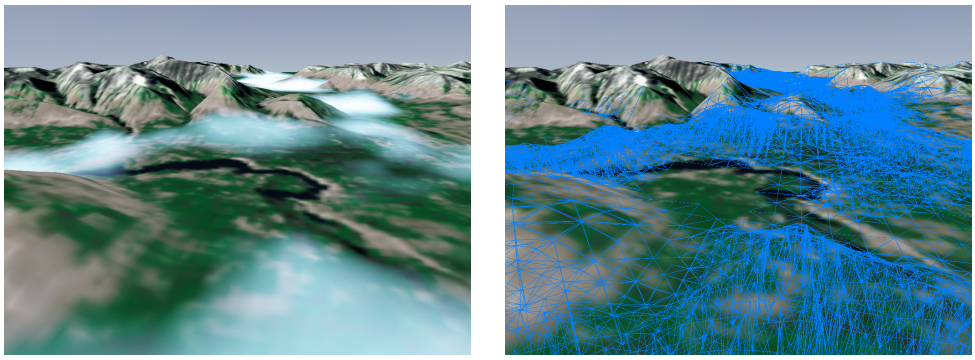


Figure 12.1: Wire frame view of ground fog which shows the view-dependent representation: Far details are represented with fewer volumetric primitives than those which are nearby.

dependent fashion. The necessity to suppress the popping effect has been addressed by a fast algorithm for volumetric morphing. We have demonstrated the performance of our algorithm by displaying non-photorealistic clouds in real time (see Chapter 11 and Figure 12.4). Because of the truly volumetric representation of the clouds, the algorithm is suited for real time weather visualization and the display of high quality volumetric effects in computer games. In order to include an application example in the area of weather visualization, Figure 12.2 depicts the simulated rising of a thunder storm cloud (data set included in the Vis5D visualization package, data courtesy of Bob Schlesinger).

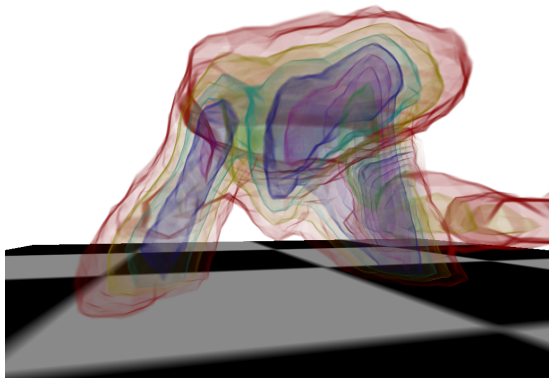


Figure 12.2: Volume visualization of a thunder storm cloud: Eight differently colored semi-transparent isosurfaces are used to visualize the shape of the storm.

12.1 Decision Chart

In order to put the existing and the new real time volume rendering algorithms in context we have included a decision chart in Figure 12.3. The chart has to be read in the following way: From top to bottom and from left to right we diversify the algorithms by means of the two criteria “quality vs. performance” and “scene complexity”, thus the appropriate algorithm for a specific rendering task can be determined quickly by considering these two criteria.

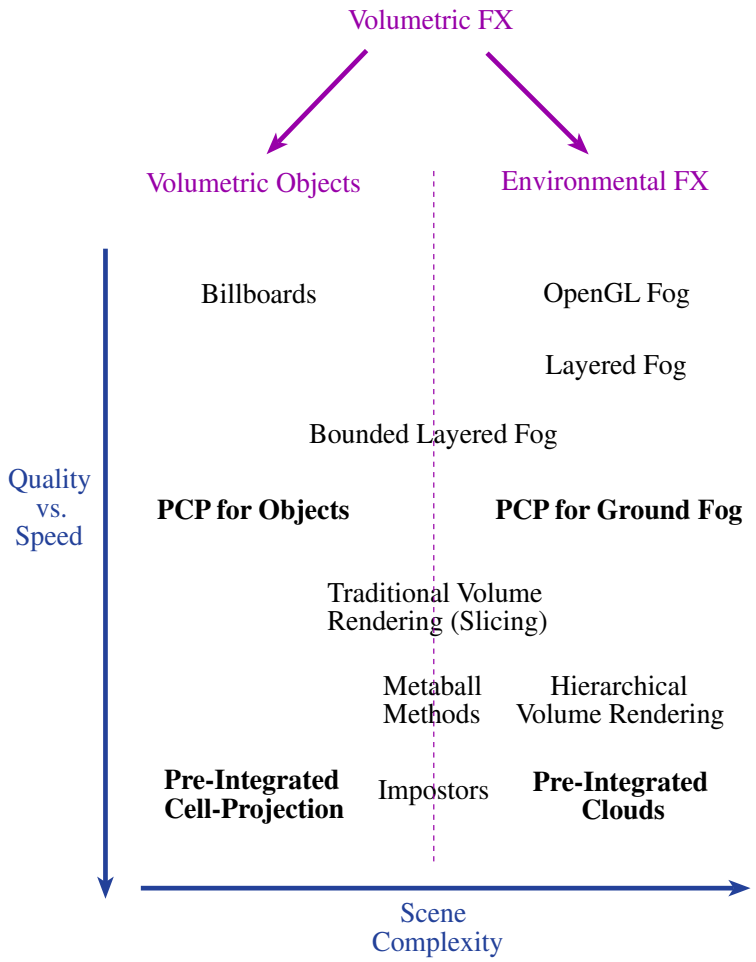


Figure 12.3: Decision chart for volumetric real time methods.

As a first decision step we take a look at the scene complexity of volumetric real time effects. We distinguish between volumetric objects like puffs of smoke and global environmental effects like fog. Naturally the scene complexity of envi-

ronmental effects is larger since the entire scene is affected. As a second step we consider the quality requirements. If a simple optical model can be used one can choose the PCP method for example. If this is not sufficient one would vote for pre-integration. The traditional volume rendering methods and the metaball method fall somewhere in between the two main categories. Because of the large rasterization overhead it is difficult to apply them to the entire scene. In that sense hierarchical volume rendering methods perform much better, but problems like temporal aliasing and limited texture memory make smooth rendering very difficult. The highest quality is achieved by pre-integration and impostor based methods. If scattering effects are required impostor methods are the best choice, but if arbitrary cloud models have to be visualized this can only be achieved with a truly volumetric method like pre-integrated cloud rendering.

In conclusion, the newly introduced unstructured volume rendering methods widen the spectrum of application areas on both the side of quality and performance. The best example for this are the new ground fog and cloud rendering methods. My hope is that some of the presented algorithms provide clues for implementing advanced volumetric FX in scientific visualization and interactive entertainment, as well. In the future this goal might be reached by establishing the tetrahedron as a basic volume rendering primitive in the API of core graphics libraries.

12.2 Availability and Licensing

The terrain and ground fog renderer referenced in Chapter 10 is available freely on the home page of the author (www9.informatik.uni-erlangen.de/Persons/Roettger/). It is licensed under the terms of the LGPL and is also part of the vtp terrain rendering package (www.vterrain.org).

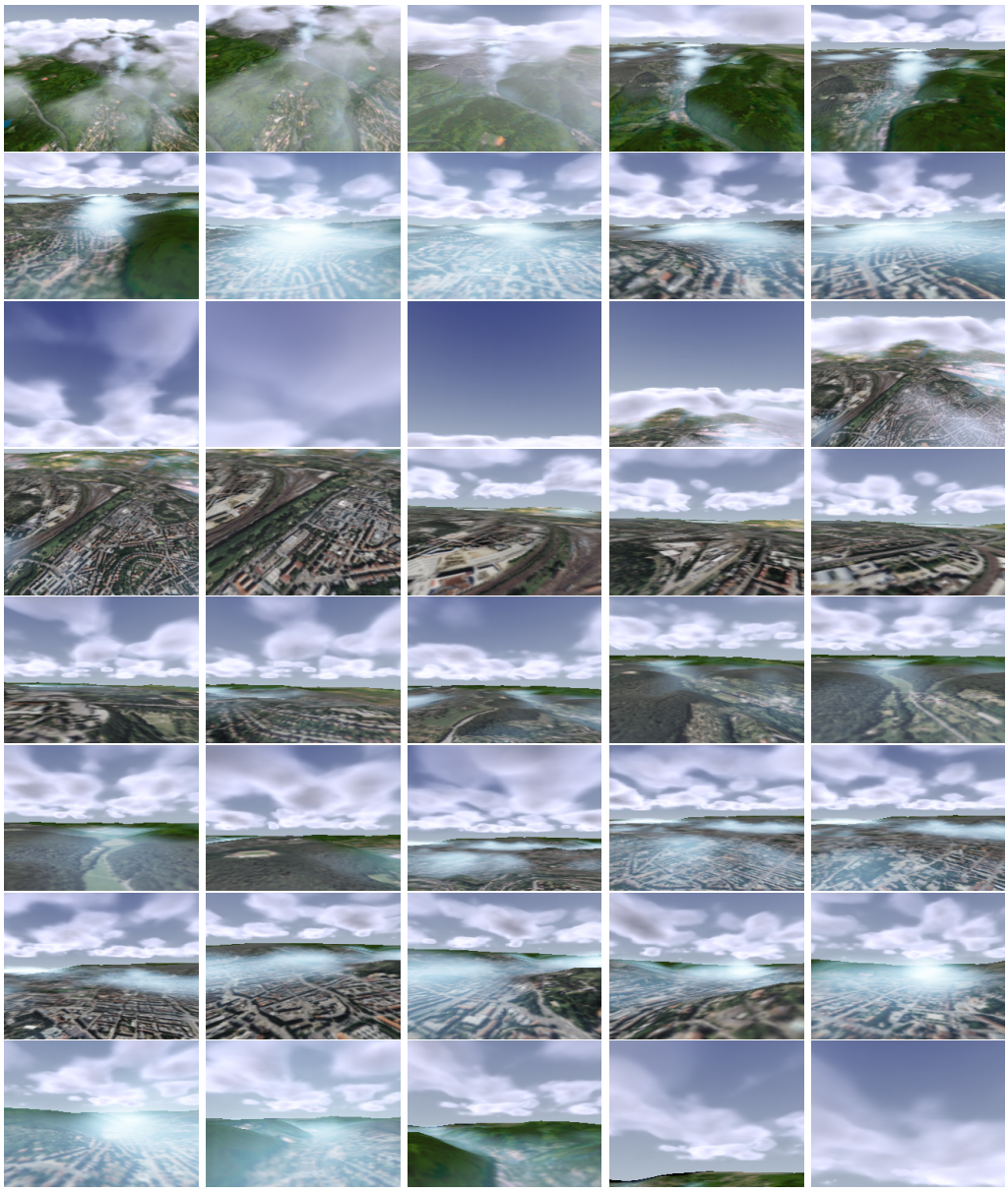


Figure 12.4: Scenes from a flight above Stuttgart showing volumetric clouds and ground fog in real time.

Bibliography

[1] AKELEY, K. RealityEngine Graphics. *Computer Graphics (SIGGRAPH '93 Proceedings) 27* (1993), 109–116.

[2] BANK, R., SHERMAN, A., AND WEISER, A. Refinement Algorithm and Data Structures for Regular Local Mesh Refinement. *Scientific Computing 44* (1983), 3–17.

[3] BLINN, J. F. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. *Computer Graphics 16*, 3 (1982), 21–29.

[4] BOADA, I., NAVAZO, I., AND SCOPIGNO, R. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer* (2001), 185–197.

[5] C. L. BAJAJ (ED.). *Data Visualization Techniques*. John Wiley & Sons, 1999.

[6] C. L. BAJAJ, V. PASCUCCI, AND D. R. SCHIKORE. Fast Isocontouring for Improved Interactivity. In *Proc. IEEE Symposium on Volume Visualization '96* (1996), pp. 39–46.

[7] CABRAL, B., CAM, N., AND FORAN, J. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proc. Symposium on Volume Visualization '94* (1994), ACM SIGGRAPH, pp. 91–98.

[8] CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND PONCHIO, F. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In *Proc. Visualization '03* (2003), IEEE, pp. 147–155.

[9] CIGNONI, P., MARINO, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 158–170.

[10] CIGNONI, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. Optimal Isosurface Extraction from Irregular Volume Data. In *Proc. IEEE Symposium on Volume Visualization '96* (1996), pp. 31–38.

[11] CIGNONI, P. AND COSTANZA, C. AND MONTANI, C. AND ROCCHINI, C. AND SCOPIGNO, R. Simplification of Tetrahedral Meshes with Accurate Error Evaluation. In *Proc. Visualization '00* (2000), IEEE, pp. 85–92.

[12] COHEN-OR, D., AND LEVANONI, Y. Temporal Continuity of Levels of Detail in Delaunay Triangulated Terrain. In *Proc. Visualization '96* (1996), IEEE Computer Society Press, pp. 37–42.

[13] COMBA, J., KLOSOWSKI, J. T., MAX, N. L., MITCHELL, J. S. B., SILVA, C. T., AND WILLIAMS, P. L. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids. *Computer Graphics Forum (Proc. Eurographics '99) 18*, 3 (1999), 369–376.

[14] CRAWFIS, R., AND MAX, N. Texture splats for 3D scalar and vector field visualization. In *Proc. Visualization '93* (1993), IEEE Computer Society, pp. 261–266.

[15] DE BOER, W. H. Fast Terrain Rendering Using Geometrical Mipmapping. *E-mersion Project* (2000).

[16] DOBASHI, Y., KANEDA, K., YAMASHITA, H., OKITA, T., AND NISHITA, T. A Simple, Efficient Method for Realistic Animation of Clouds. In *Proc. SIGGRAPH '00* (2000), ACM, pp. 19–28.

[17] DOBASHI, Y., YAMAMOTO, T., AND NISHITA, T. Interactive Rendering of Atmospheric Scattering Effects Using Graphics Hardware. In *Proc. EG/SIGGRAPH Graphics Hardware Workshop '02* (2002), pp. 99–108.

[18] DREBIN, R. A., CARPENTER, L., AND HANRAHAN, P. Volume Rendering. *Computer Graphics 22*, 4 (1988), 65–74.

[19] DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. ROAMing Terrain: Real-Time Optimally Adapting Meshes. In *Proc. Visualization '97* (1997), IEEE, pp. 81–88.

[20] DURKIN, J. W., AND HUGHES, J. F. Nonpolygonal Isosurface Rendering for Large Volume Datasets. In *Proc. IEEE Visualization '94* (1994), pp. 293–300.

[21] E. W. WEISSTEIN. *Eric Weisstein's World Of Mathematics.* `http://mathworld.wolfram.com/`, 2004.

[22] EBERT, D., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing & Modeling, A Procedural Approach*, second edition, isbn 0-12-228730-4 ed. AP Professional, 1998.

[23] EBERT, D., AND PARENT, R. Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-Buffer Techniques. *Computer Graphics (Proc. SIGGRAPH '90) 24*, 4 (1990), 357–366.

[24] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution Analysis of Arbitrary Meshes. In *Proc. SIGGRAPH '95* (1995), ACM, pp. 173–182.

[25] ELINAS, P., AND STUERZLINGER, W. Real-time Rendering of 3D Clouds. *Journal of Graphics Tools 5*, 4 (2000), 33–45.

[26] ENGEL, K., KRAUS, M., AND ERTL, T. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics Workshop on Graphics Hardware '01* (2001), ACM SIGGRAPH, pp. 9–16.

[27] ENGEL, K., WESTERMANN, R., AND ERTL, T. Isosurface Extraction Techniques for Web-Based Volume Visualization. In *Proc. IEEE Visualization '99* (1999), pp. 139–146.

[28] FARIAS, R., MITCHELL, J., AND SILVA, C. An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In *Proc. IEEE Symposium on Volume Visualization '00* (2000), ACM Press, pp. 91–99.

[29] FEDKIW, R., STAM, J., AND JENSEN, H. W. Visual Simulation of Smoke. In *Proc. SIGGRAPH '01* (2001), ACM, pp. 15–22.

[30] FOWLER, R. J., AND LITTLE, J. J. Automatic Extraction of Irregular Network Digital Terrain Models. *Computer Graphics 13*, 2 (1979), 199–207.

[31] GARDNER, G. Y. Visual Simulation of Clouds. In *Proc. SIGGRAPH '85* (1985), ACM, pp. 297–303.

[32] GERSTNER, T. AND MEETSCHEN, D. AND CREWELL, S. AND GRIEBEL, M. AND SIMMER, C. A Case Study on Multiresolution Visualization of Local Rainfall from Weather Radar Measurements. In *Proc. IEEE Visualization '02* (2002), pp. 533–536.

[33] GROSS, M. H., GATTI, R., AND STAADT, O. Fast Multiresolution Surface Meshing. In *Proc. Visualization '95* (1995), IEEE Computer Society Press, pp. 135–142.

[34] GROSSO, R., LÜRIG, C., AND ERTL, T. The Multilevel Finite Element Method for Adaptive Mesh Optimization and Visualization of Volume Data. In *Proc. Visualization '97* (1997), IEEE, pp. 387–394.

[35] GUTHE, S., ROETTGER, S., SCHIEBER, A., STRASSER, W., AND ERTL, T. High-Quality Unstructured Volume Rendering on the PC Platform. In *Proc. EG/SIGGRAPH Graphics Hardware Workshop '02* (2002), pp. 119–125.

[36] GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. Interactive Rendering of Large Volume Data Sets. In *Proc. Visualization '02* (2002), IEEE Computer Society Press, pp. 53–60.

[37] HARRIS, M. J., AND LASTRA, A. Real-Time Cloud Rendering. *Computer Graphics Forum (Proc. Eurographics '01) 20*, 3 (2001), 76–84.

[38] HEIDRICH, W., MCCOOL, M., AND STEVENS, J. Interactive Maximum Projection Volume Rendering. In *Proc. Visualization '95* (1995), IEEE, pp. 11–18.

[39] HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. In *Proc. ACM Symposium on Interactive 3D Graphics* (1999), pp. 127–134.

[40] HOPPE, H. Progressive meshes. In *Computer Graphics (Proceedings SIGGRAPH '96)* (1996), pp. 99–108.

[41] HOPPE, H. View-dependent refinement of progressive meshes. In *Computer Graphics (Proceedings SIGGRAPH '97)* (1997), pp. 189–198.

[42] HOPPE, H. Smooth View-Dependant Level-of-Detail Control and its Application to Terrain Rendering. In *Proc. Visualization '98* (1998), IEEE, pp. 35–42.

[43] JENSEN, H. W., AND CHRISTENSEN, P. H. Efficient Simulation of Light Transport in Scenes with Participating Media Using Photon Maps. In *Proc. SIGGRAPH '98* (1998), ACM, pp. 311–320.

[44] KAJIYA, J. T. The Rendering Equation. *Computer Graphics (ACM SIGGRAPH '86 Proceedings) 20* (1986), 143–150.

[45] KAJIYA, J. T., AND VON HERZEN, B. P. Ray Tracing Volume Densities. *Computer Graphics (ACM SIGGRAPH '84 Proceedings) 18*, 3 (1984), 165–174.

[46] KANITSAR, A., THEUSSL, T., MROZ, L., SRAMEK, M., VILANOVA, A., CSEBFALVI, B., HLADUVKA, J., FLEISCHMANN, D., KNAPP, M., WEGENKITTL, R., FELKEL, P., ROETTGER, S., GUTHE, S., PURGATHOFER, W., AND E., G. Christmas Tree Case Study: Computed Tomography as a Tool for Mastering Complex Real World Objects with Applications in Computer Graphics. In *Proc. Visualization '02* (2002), IEEE Computer Society Press, pp. 489–492.

[47] KING, D., WITTENBRINK, C., AND WOLTERS, H. An Architecture For Interactive Tetrahedral Volume Rendering. *Proc. International Workshop on Volume Graphics '01* (2001), 101–112.

[48] KNISS, J., PREMOZE, S., HANSEN, C., SHIRLEY, P., AND MCPHERSON, A. A Model for Volume Lighting and Modeling. In *Transactions on Visualization and Computer Graphics 2003* (2003), pp. 150–162.

[49] KNISS, J., PREMOZE, S., IKITS, M., LEFOHN, A., HANSEN, C., AND PRAUN, E. Gaussian Transfer Functions for Multi-Field Volume Visualization. In *Proc. Visualization '03* (2003), pp. 497–504.

[50] KOLLER, D., LINDSTROM, P., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. Virtual GIS: A real-time 3D geographic information system. In *Proc. Visualization '95* (1996), IEEE Computer Society Press, pp. 94–100.

[51] KRAUS, M., AND ERTL, T. Cell-Projection of Cyclic Meshes. In *Proc. IEEE Visualization '01* (2001), pp. 215–222.

[52] KREEGER, K., AND KAUFMAN, A. Mixing Translucent Polygons with Volumes. In *Proc. IEEE Visualization '99* (1999), pp. 191–198.

[53] LACROUTE, P. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proc. Parallel Rendering Symposium '95* (1995), pp. 15–22.

[54] LACROUTE, P., AND LEVOY, M. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Computer Graphics (Proceedings SIGGRAPH '94) 28*, 4 (1994), 451–457.

[55] LAMAR, E. C., HAMANN, B., AND JOY, K. I. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proc. Visualization '99* (1999), IEEE, pp. 355–362.

[56] LAUR, D., AND HANRAHAN, P. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proc. SIGGRAPH '91* (1991), pp. 285–288.

[57] LEGAKIS, J. Fast Multi-Layer Fog. In *ACM SIGGRAPH '98 Conference Abstracts and Applications* (1998), p. 266.

[58] LIN, C.-C., AND CHING, Y.-T. An Efficient Volume-Rendering Algorithm with an Analytic Approach. *The Visual Computer 12*, 10 (1996), 515–526.

[59] LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. Real-Time, Continuous Level of Detail Rendering of Height Fields. In *Proc. SIGGRAPH '96* (1996), ACM, pp. 109–118.

[60] LIVNAT, Y., AND HANSEN, C. View Dependent Isosurface Extraction. In *Proc. IEEE Visualization '98* (1998), pp. 175–180.

[61] LIVNAT, Y., SHEN, H.-W., AND JOHNSON, C. R. A Near Optimal Isosurface Extraction Algorithm Using Span Space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84.

[62] LORENSEN, W. E., AND CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics (Proc. SIGGRAPH '87) 21*, 4 (1987), 163–169.

[63] M. LEVOY. Display of Surfaces from Volume Data. *Computer Graphics and Applications 8*, 3 (1988), 29–37.

[64] MALZBENDER, T., GELB, D., AND WOLTERS, H. Polynomial Texture Maps. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Annual Conference Series, pp. 519–528.

[65] MASSIVE DEVELOPMENT. Krass Game Engine. *http://www.massive.de/english/technology_eng.html* (2000).

[66] MAX, N., BECKER, B., AND CRAWFIS, R. Flow Volumes for Interactive Vector Field Visualization. In *Proc. Visualization '93* (1993), IEEE Computer Society Press, pp. 19–24.

[67] MAX, N. L. Efficient Light Propagation for Multiple Anisotropic Volume Scattering. In *5th Workshop on Rendering* (1994), Eurographics, pp. 87–104.

[68] MAX, N. L. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 99–108.

[69] MAX, N. L., HANRAHAN, P., AND CRAWFIS, R. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics (San Diego Workshop on Volume Visualization) 24*, 5 (1990), 27–33.

[70] MECH, R. Hardware-Accelerated Real-Time Rendering of Gaseous Phenomena. *Journal of Graphics Tools 6*, 3 (2001), 1–16.

[71] MEISSNER, M., GUTHE, S., AND STRASSER, W. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *Proc. Graphics Interface '02* (2002), pp. 209–218.

[72] MEISSNER, M., KANUS, U., WETEKAM, G., HIRCHE, J., EHLERT, A., STRASSER, W., DOGGETT, M., FORTHMANN, P., AND PROKSA, R. VIZARDII: A Reconfigurable Interactive Volume Rendering System. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware '02* (2002), pp. 137–146.

[73] MEREDITH, J., AND MA, K. Multi-Resolution View-Dependent Splat Based Volume Rendering of Large Irregular Data. In *Proc. Symposium on Large-Data Visualization and Graphics '01* (2001), pp. 93–99.

[74] MITCHELL, J. L. 1.4 Pixel Shaders. *Meltdown* (2001).

[75] MIYAZAKI, R., YOSHIDA, S., DOBASHI, Y., AND NISHITA, T. A Method for Modeling Clouds Based on Atmospheric Fluid Dynamics. In *Proc. Pacific Graphics '01* (2001), pp. 363–372.

[76] MONTANI, C., SCATENI, R., AND SCOPIGNO, R. Discretized Marching Cubes. In *Proc. IEEE Visualization '94* (1994), pp. 281–287.

[77] NEYRET, F. Qualitative Simulation of Cloud Formation and Evolution. In *8th Workshop on Computer Animation and Simulation (EGCAS '97)* (Wien, 1997), Eurographics, Springer, pp. 113–124.

[78] NISHITA, T., DOBASHI, Y., AND NAKAMAE, E. Display of Clouds Taking into Account Multiple Anisotropic Scattering and Sky Light. In *Proc. SIGGRAPH '96* (1996), ACM, pp. 379–386.

[79] OH, K.-M., AND PARK, K. H. A Type-Merging Algorithm for Extracting an Isosurface from Volumetric Data. *The Visual Computer 12*, 8 (1996), 406–419.

[80] OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL Reference Manual*. Addison-Wesley, 1992.

[81] PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., , AND SLOAN, P.-P. Interactive Ray Tracing for Isosurface Rendering. In *Proc. IEEE Visualization '98* (1998), pp. 233–238.

[82] PERLIN, K. An Image Synthesizer. *Computer Graphics (Proc. SIGGRAPH '85) 19*, 3 (1985), 287–296.

[83] PFISTER, H., HARDENBERGH, J., KNITTEL, J., LAUER, H., AND SEILER, L. The VolumePro Real-Time Ray-Casting System. In *Proc. SIGGRAPH '99* (1999), pp. 251–260.

[84] ROETTGER, S., AND ERTL, T. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proc. IEEE Symposium on Volume Visualization '02* (2002), ACM Press, pp. 23–28.

[85] ROETTGER, S., AND ERTL, T. Cell Projection of Convex Polyhedra. In *Proc. Volume Graphics '03* (2003), pp. 103–107.

[86] ROETTGER, S., AND ERTL, T. Fast Volumetric Display of Natural Gaseous Phenonema. In *Proc. Computer Graphics International '03* (2003), pp. 74–81.

[87] ROETTGER, S., AND FRICK, I. The Terrain Rendering Pipeline. In *Proc. EWV '02* (2002), OCG Schriftenreihe, R. Oldenburg, Vienna, pp. 195–199.

[88] ROETTGER, S., GUTHE, S., WEISKOPF, D., AND ERTL, T. Smart Hardware-Accelerated Volume Rendering. In *Proc. Visualization Symposium '03* (2003), IEEE Computer Society Press, pp. 231–238.

[89] ROETTGER, S., HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Proc. WSCG '98* (1998), EG/IFIP, pp. 315–322.

[90] ROETTGER, S., KRAUS, M., AND ERTL, T. Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection. In *Proc. Visualization '00* (2000), IEEE, pp. 109–116.

[91] ROSSIGNAC, J., AND BORREL, P. *Multi-Resolution 3D Approximations for Rendering*. Springer Verlag, 1993, pp. 455–465.

[92] RUSHMEIER, H. *Realistic Image Synthesis for Scenes with Radiatively participating Media*. Cornell University, 1988.

[93] RUSHMEIER, H., AND TORRANCE, K. The Zonal Method for Calculating Light Intensities in the Presence of Participating Media. *Computer Graphics 21*, 4 (1987), 293–302.

[94] SAMET, H. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[95] SCHAUFLER, G. Per-Object Image Warping with Layered Impostors. In *Proc. 9th Workshop on Rendering '98* (1998), Eurographics, pp. 145–156.

[96] SCHUSSMAN, G., AND MAX, N. L. Hierarchical Perspective Volume Rendering Using Triangle Fans. In *Proc. TCVG Eurographics Workshop (VolumeGraphics '01)* (2001), IEEE/EG, pp. 309–320.

[97] SHADE, J., GORTLER, S., HE, L., AND SZELISKI, R. Layered Depth Images. In *Proc. SIGGRAPH '98* (1998), ACM, pp. 231–242.

[98] SHEKHAR, R., FAYYAD, E., YAGEL, R., AND CORNHILL, J. F. Octree-Based Decimation of Marching Cubes Surfaces. In *Proc. IEEE Visualization '96* (1996), pp. 335–342.

[99] SHEN, H.-W., HANSEN, C. D., LIVNAT, Y., AND JOHNSON, C. R. Isosurfacing in Span Space with Utmost Efficiency (ISSUE). In *Proc. IEEE Visualization '96* (1996), pp. 287–294.

[100] SHEN, H.-W., AND JOHNSON, C. R. Sweeping Simplices: A Fast Iso-Surface Extraction Algorithm for Unstructured Grids. In *Proc. IEEE Visualization '95* (1995), pp. 143–150.

[101] SHIRLEY, P., AND TUCHMAN, A. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics (San Diego Workshop on Volume Visualization) 24*, 5 (1990), 63–70.

[102] SIEGEL, R., AND HOWELL, J. *Thermal Radiation Heat Transfer*. McGraw Hill, New York, 1981.

[103] SILVA, C. T., AND MITCHELL, J. S. B. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 142–157.

[104] SILVA, C. T., MITCHELL, J. S. B., AND WILLIAMS, P. L. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *Proc. IEEE Symposium on Volume Visualization '98* (1998), ACM Press, pp. 87–94.

[105] STAM, J., AND FIUME, E. Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes. In *Proc. SIGGRAPH '95* (1995), ACM, pp. 129–136.

[106] STEIN, C. M., BECKER, B. G., AND MAX, N. L. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proc. IEEE Symposium on Volume Visualization '94* (1994), pp. 83–89.

[107] SUTER, M., AND NÜESCH, D. Automated generation of visual simulation databases using remote sensing and GIS. In *Proc. Visualization '95* (1995), IEEE Computer Society Press, pp. 135–142.

[108] ŠRÁMEK, M. Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance. In *Proc. IEEE Visualization '94* (1994), pp. 188–195.

[109] WEILER, M., AND ERTL, T. Hardware-Based View-Independent Cell Projection. In *Proc. IEEE Symposium on Volume Visualization '02* (2002), ACM Press, pp. 13–22.

[110] WEILER, M., KRAUS, M., MERZ, M., AND ERTL, T. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. Visualization '03* (2003), IEEE, pp. 333–340.

[111] WEILER, M., WESTERMANN, R., HANSEN, C., ZIMMERMAN, K., AND ERTL, T. Level-Of-Detail Volume Rendering via 3D Textures. In *Volume Visualization and Graphics Sympsium '00* (2000), IEEE, pp. 7–13.

[112] WESTERMANN, R., AND ERTL, T. The VSBUFFER: Visibility Ordering of Unstructured Volume Primitives by Polygon Drawing. In *Proc. Visualization '97* (1997), IEEE, pp. 35–42.

[113] WESTERMANN, R., AND ERTL, T. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics* (1998), Annual Conference Series, ACM, pp. 169–177.

[114] WESTERMANN, R., JOHNSON, C. R., AND ERTL, T. A Level-Set Method for Flow Visualization. In *Proc. IEEE Visualization '00* (2000), pp. 147–154.

[115] WESTOVER, L. Interactive Volume Rendering. In *Proc. Chapel Hill Workshop on Volume Visualization* (1989), pp. 9–16.

[116] WILHELMS, J., AND VAN GELDER, A. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics 25*, 4 (1991), 275–284.

[117] WILHELMS, J., AND VAN GELDER, A. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics 11*, 2 (1992), 201–227.

[118] WILLIAMS, P. L. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics 11*, 2 (1992), 103–126.

[119] WILLIAMS, P. L., AND MAX, N. L. A Volume Density Optical Model. In *Computer Graphics (Workshop on Volume Visualization '92)* (1992), ACM, pp. 61–68.

[120] WILLIAMS, P. L., MAX, N. L., AND STEIN, C. M. A High Accuracy Volume Renderer for Unstructured Data. *Transactions on Visualization and Computer Graphics 4*, 1 (1998), 37–54.

[121] WITTENBRINK, C. M. CellFast: Interactive Unstructured Volume Rendering. In *IEEE Visualization '99 Late Breaking Hot Topics* (1999), pp. 21–24.

[122] WOLFGANG HEIDRICH. *High-quality Shading and Lighting for Hardware-accelerated Rendering.* PhD thesis, University of Erlangen-Nuremberg, 1999.

[123] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd ed.* Addison-Wesley, 1999.

[124] WYLIE, B., MORELAND, K., FISK, L. A., AND CROSSNO, P. Tetrahedral Projection using Vertex Shaders. In *Proc. IEEE Symposium on Volume Visualization '02* (2002), ACM Press, pp. 7–12.

[125] WYVILL, B., WYVILL, G., AND MCPHEETERS, C. Data Structures for Soft Objects. *The Visual Computer 2* (1986), 227–234.

[126] XIA, J. C., EL-SANA, J., AND VARSHNEY, A. Adaptive Real-Time Level-of-Detail Based Rendering for Polygonal Models. *Trans. on Visualization and Computer Graphics 3*, 2 (1997), 171–183.

[127] YAGEL, R., REED, D. M., LAW, A., SHIH, P., AND SHAREEF, N. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In *Proc. IEEE Symposium on Volume Visualization '96* (1996), pp. 55–62.

[128] ZHANG, H. Vertex Program 1.1 and Texture Shader 3. *Game Developers Conference* (2002).

[129] ZHOU, Y., CHEN, B., AND KAUFMAN, A. Multiresolution Tetrahedral Framework for Visualizing Regular Volume Data. In *Proc. Visualization '97* (1997), IEEE, pp. 135–142.

List of Figures

List of Figures **5**

2.1 The OpenGL rendering pipeline 12
2.2 An OpenGL rendering example 14
2.3 The OpenGL rendering primitives 15
2.4 Lighting and texturing 16
2.5 Light reflection 17
2.6 BRDF 18
2.7 Texture mapping 19
2.8 MIP-mapping 20
2.9 3D-Texturing 21
2.10 Register combiners 23

3.1 Triangular irregular network 27
3.2 Static LOD as proposed by Koller et al. 29
3.3 Progressive meshes 30
3.4 C-LOD algorithm as proposed by Lindstrom et al. 31
3.5 Projection of a delta segment 32
3.6 Example of ROAM terrain 33
3.7 Split and merge operation 34
3.8 C-LOD as proposed by Roettger et al. 36
3.9 Terrain quadtree 37

4.1 Example screen shot of AquaNox 40
4.2 The main stages of the terrain rendering pipeline 45

5.1 Sky dome 48
5.2 Absorption and scattering 49
5.3 OpenGL fog 49
5.4 Layered fog 50
5.5 Bounded layered fog 52
5.6 Metaball clouds 53

5.7 Impostor clouds 55
5.8 Dobashi's method 56
5.9 Harris' method 56

6.1 Multiple scattering 61
6.2 Anisotropic scattering 62

7.1 Basic principle of ray casting 65
7.2 Hardware-accelerated volume rendering 66

8.1 Classification of tetrahedra according to PT algorithm 69
8.2 Viewing ray intersecting a tetrahedron 71
8.3 3D pre-integration table 73
8.4 2D pre-integration table 73
8.5 Isosurface rendering 76
8.6 Isosurface classification 78
8.7 Weighting coefficients for gradient interpolation 80
8.8 Front face texture map 81
8.9 Back face texture map 81
8.10 Example of multiple isosurfaces 82
8.11 Mixing isosurfaces with projected volumes 82
8.12 Example 2D pre-integration textures 85
8.13 Bluntfin data set with multiple isosurfaces 86
8.14 MRI head scan 86
8.15 CT bonsai scan 86
8.16 Pre- vs. post-classification 87

9.1 Comparison between linear α and exponential approximation. . . 89
9.2 Comparison between linear and quadratic color approximation . . 92
9.3 Hardware-accelerated pre-integration 93
9.4 Comparison between hardware and software pre-integration . . . 95
9.5 Comparison between different approximations of the ray integral . 97
9.6 Bucky Ball with per-vertex lighting 97
9.7 Blunt Fin dataset using quadratic polynomial approximation . . . 97

10.1 Projection of polyhedral cells 101
10.2 Blunt Fin and Bucky Ball data set 102
10.3 Synthetic data sets 103
10.4 Timings for hexahedral projection 103
10.5 Stacking prisms onto a triangulated surface 104
10.6 Ground fog generated with 2D Perlin noise 105

11.1 Hierarchical volume representation using an octree 108
11.2 Two-dimensional morphing example 112
11.3 The city center of Stuttgart 115
11.4 The impact of different transfer functions 116
11.5 Real and virtual panorama of Stuttgart 117

12.1 Wire frame view of ground fog 119
12.2 Volume visualization of a thunder storm cloud 119
12.3 Decision chart 120
12.4 Scenes from a flight above Stuttgart 122

List of Tables

List of Tables **5**

8.1 Rendering times for isosurfaces 84

9.1 Texture setup for dependent texture mapping 90
9.2 Texture setup for polynomial color approximation 91
9.3 1D texture used for hardware-accelerated pre-integration 94
9.4 Preprocessing times for 2D multi-texturing 96
9.5 Display times on a Pentium 4 (2 GHz) 96

Chapter 13

Zusammenfassung

Das Ziel dieser Dissertation ist die Entwicklung von volumetrischen Methoden zur Darstellung von natürlichen Phänomenen wie zum Beispiel Wolken und Bodennebel. Dies ist besonders in Computerspielen wichtig, in denen die stetige Weiterentwicklung der in Personalcomputern eingesetzten Graphikkarten die realistische und gleichzeitig echtzeitfähige Darstellung von dreidimensionalen Szenen ermöglicht haben.

Nach einer kurzen Motivation und Einführung in dreidimensionale Visualisierungstechniken folgt in Kapitel 3 ein Überblick über die Arbeitsweise der Graphikhardware. In Kapitel 4 werden Methoden zur Darstellung von Landschaften behandelt, welche die Basis für jede Outdoor-Game-Engine darstellen. Im Verlauf der Dissertation werden wir mehrere Visualisierungsalgorithmen vorstellen, die auf diesen Darstellungsmethoden aufbauen. Insbesondere die so genannten C-LOD Algorithmen werden ausführlich behandelt und miteinander verglichen, da sie zur Zeit die am weitesten fortgeschrittene Technik darstellen.

Die Notwendigkeit, fortgeschrittene Algorithmen zur Landschaftsvisualisierung zu verwenden, liegt in der schieren Größe der Daten begründet. Eine Landschaft wird im Allgemeinen durch ein so genanntes Höhenfeld beschrieben. Dies ist eine zwei-dimensionale Matrix, welche die Höhenwerte enthält. Die durchschnittliche Größe dieses Höhenfeldes liegt heutzutage bei 2000x2000 bis 4000x4000 Gitterpunkten. Aktuelle kommerziell genutzte Satelliten liefern aber bereits eine Auflösung von unter 10 Metern. Militärsatelliten erreichen jedoch schon seit geraumer Zeit eine Auflösung von deutlich unter einem Meter. So erhält man für eine Auflösung von nur einem Kilometer bereits eine Datenmenge von rund 500 Millionen Gitterpunkten für die gesamte Erde. Für eine Auflösung von 10 Metern erhöht sich die Datenmenge entsprechend auf 5 Billionen Gitterpunkte. Das entspricht einer unkomprimierten Datenmenge von rund 10 Terabyte. Dies verdeutlicht, welche Datenmenge bei der Landschaftsvisualisierung theoretisch pro Bild verarbeitet werden muss. Selbst bei der erwähnten durchschnittlichen Größe von 2000x2000 Punkten besteht die Landschaft noch aus 4 Millionen Dreiecken. Unter der Annahme, dass aktuelle Graphikkarten circa 30 Millionen Dreiecke pro Sekunde verarbeiten können, ergibt sich eine Bildwiederholrate von ca. 7 Bildern pro Sekunde. Für interaktive Anwendungen ist dies bei weitem nicht

schnell genug. Man verwendet daher Algorithmen die nicht die gesamte Menge an Dreiecken zeichnen, sondern nur diejenigen, die für den korrekten visuellen Eindruck notwendig sind. So kann man sich zum Beispiel vorstellen, dass eine kleine Erhebung auf ansonsten glatter Oberfläche in weiter Entfernung kleiner als ein Pixel dargestellt werden würde. Dieses Detail kann problemlos vernachlässigt werden, ohne den Gesamteindruck zu verfälschen. Genau dies ist das Prinzip, welches den C-LOD Algorithmen zugrunde liegt.

In den darauf folgenden Kapiteln 5 und 6 erfolgt eine Bestandsaufnahme, welche volumetrischen Techniken aktuell in Computerspielen eingesetzt werden. Dazu wird die Rendering-Engine des aktuellen Computerspiels AquaNox von Massive Development unter die Lupe genommen, da sie dem aktuellen Stand der Technik entspricht. Durch die Zusammenarbeit mit Massive Development hatten wir außerdem die Gelegenheit die Interna der Game-Engine sehr gut kennen zulernen.

Der Teil der Game-Engine, der sich mit der Darstellung von virtuellen Landschaften beschäftigt, kann anhand der so genannten “Terrain Rendering Pipeline” beschrieben werden. D.h., während der Darstellung der synthetischen Landschaft durchläuft diese mehrere Phasen, in denen sie schrittweise ihr endgültiges Aussehen erhält. Diese Phasen sind insbesondere: Generierung der Geometrie, Texturierung, Beleuchtung, Platzierung von organischen Objekten und schließlich die Anwendung volumetrischer Effekte. Letzteres zu verbessern, ist das Hauptziel dieser Dissertation.

Um volumetrische Effekte zu erzielen, werden in der letzten Stufe der Pipeline hauptsächlich nur Billboard- und Layered-Fog-Techniken eingesetzt. Aber auch andere Computerspiele gehen kaum über diesen Quasi-Standard hinaus. Der Grund dafür liegt hauptsächlich darin, dass alle visuellen Effekte in der kurzlebigen Spielebranche innerhalb kürzester Zeit realisiert werden müssen. Obwohl die erwähnten Techniken nicht unbedingt realitätsgetreu zu nennen sind, sind sie wegen der einfachen Implementierbarkeit sehr beliebt. Abschließend kann man sagen, dass die volumetrischen Effekte in der Spielebranche sehr unterentwickelt sind im direkten Vergleich mit dem wissenschaftlichen Status Quo. Unser Ziel ist es nun, bekannte Volumenvisualisierungsmethoden aufzuzeigen bzw. existierende Methoden so abzuwandeln und zu verbessern, dass sie für Computerspiele geeignet sind. Als Nebeneffekt davon tragen die in der Dissertation entwickelten Methoden auch zur qualitativ besseren Darstellung in der wissenschaftlichen Volumenvisualisierung bei.

In Kapitel 6 werden nun die existierenden volumetrischen Methoden beschrieben, die generell für Computerspiele geeignet sind. Da eine Hauptvoraussetzung die interaktive Geschwindigkeit ist, schränkt das die Auswahl an Algorithmen deutlich ein. Diejenigen Methoden, die diese Voraussetzung erfüllen, sind insbesondere Sky Domes, OpenGL Fog, Layered Fog, Bounded Layered Fog, Billboards, und Impostors. Die Vor- und Nachteile jeder unterschiedlichen Methode

werden entsprechend beleuchtet. Nach dieser Bestandsaufnahme kommt man unweigerlich zu der Schlussfolgerung, dass die vorgestellten Methoden die eigentliche Aufgabe, nämlich das Strahlintegral zu berechnen, mit mehr oder weniger Geschick und Erfolg versuchen zu umgehen. Dies ist nicht verwunderlich, denn für jeden Sehstrahl, der ein Volumen durchdringt, müsste man die Abschwächung der Lichtintensität mit Hilfe dieses Strahlintegrals berechnen. Prinzip bedingt ist dies nur durch Abtastung und numerische Integration eines jeden Sehstrahls zu bewerkstelligen. Für volumetrische Effekte in Computerspielen wird nun einerseits entweder die Lösung des Strahlintegrals durch unrealistische Annahmen soweit vereinfacht, dass die Integration analytisch berechnet werden kann, oder es wird gar nicht erst versucht das Strahlintegral zu lösen. Anstelle dessen werden volumetrische Effekte häufig nur vorgetäuscht. Insgesamt kann man sagen, dass die bekannten Algorithmen aus den genannten Gründen entweder unrealistische Bilder erzeugen oder bezüglich der darstellbaren Effekte sehr eingeschränkt sind.

Um diesen Misstand zu beseitigen, versuchen wir bekannte Volumenvisualisierungsmethoden soweit zu verbessern, dass sie auch für den Einsatz in Computerspielen schnell genug sind. Eine Methode, die hier besonders viel versprechende Ergebnisse erwarten lässt, ist die Visualisierung von Volumendaten basierend auf so genannten unstrukturierten Gittern. Diese werden hauptsächlich in Finite-Elemente-Simulationen eingesetzt, da dadurch komplizierte Simulationsgeometrien mit vergleichsweise wenigen Gitterelementen beschrieben werden können. Die Haupteinsatzbereiche sind hierbei, um nur einige Beispiele zu nennen, die Strömungssimulation in der Luftfahrt- und Automobilindustrie, oder der Wärmetransport in komplexen technischen Baugruppen. Je weniger Elemente das Simulationsgitter aufweist, desto schneller ist auch die Simulation und letztendlich auch die Visualisierung. Die Herangehensweise mittels unstrukturierter Gitter ist auch für Computerspiele vorteilhaft, da ein Großteil der volumetrischen Effekte effizient auf unstrukturierte Gitter abgebildet werden kann.

Ein erster Einblick in die Volumenvisualisierung von unstrukturierten Gittern wird in den Kapiteln 7 bis 10 gegeben. Kapitel 7 und 8 behandeln generell die Grundlagen der Volumenvisualisierung speziell im Hinblick auf die Darstellung von natürlichen Phänomenen wie z.B. Wolken. Es werden hauptsächlich die physikalischen Mechanismen des Strahlungstransports in gasförmigen Medien behandelt, was physikalische Effekte wie Emission, Abschwächung und Streuung mit einschließt.

Kapitel 9 beschreibt, wie man das Strahlintegral exakt und effizient lösen kann, wenn man die Standardmethode zur Visualisierung unstrukturierter Gitter, den PT-Algorithmus, verwendet. Der PT-Algorithmus, der besser unter dem Begriff Zellprojektion bekannt ist, projiziert einen Tetraeder in die Bildebene, so dass der Tetraeder mit Graphikprimitiven, wie sie jede aktuelle Graphikkarte bietet, dargestellt werden kann. Aufgrund der hardwarebeschleunigten Darstellungswei-

se ist das Verfahren sehr effizient. Da es ein Objektraumverfahren ist, hängt die Performanz hauptsächlich von der Anzahl der Tetraeder ab und nicht wie beim Ray-Tracing oder Ray-Casting von der Anzahl der Bildpunkte.

Ein Nachteil des ursprünglichen PT-Algorithmus war, dass bis dato nur Prä-Klassifikation möglich war, was sich darin manifestierte, dass die Transferfunktion innerhalb eines Tetraeders nicht korrekt wiedergegeben werden konnte. Da im allgemeinen Volumendaten in Form von Skalarwerten gegeben sind, muss jedem Skalarwert eindeutig eine optische Dichte zugewiesen werden. Dies geschieht mit einer Transferfunktion. Außerdem legt die Transferfunktion auch die Eigenemission des gasförmigen Mediums fest. Unter Zuhilfenahme der Fähigkeiten moderner Graphikhardware konnten wir zeigen, dass mittels der so genannten Prä-Integrationsmethode eine per-Pixel exakte Darstellung der Tetraeder für beliebige Transferfunktionen erzielt werden kann. Dies hat eine deutliche Qualitätssteigerung zur Folge und stellt einen fundamentalen Fortschritt im Vergleich mit den bisher bekannten Methoden dar (vergleiche Abbildung 8.16).

Dies wurde durch eine geschickte Parametrisierung des Strahlintegrals erreicht. Jedes Segment eines Sichtstrahls durch einen Tetraeder kann durch drei Parameter eindeutig beschrieben werden: Den Skalarwert am Eintritts- und Austrittspunkt des Sichtstrahls und die Länge des Strahlsegments. Dadurch ist auch das zum Strahlsegment gehörige Strahlintegral eindeutig definiert. Indem man nun das Strahlintegral für jede Kombination der drei Parameter numerisch vorberechnet und in einer drei-dimensionalen Tabelle speichert, ist das exakte Strahlintegral während des Zeichnens eines Tetraeders schnell verfügbar. Man kann sogar soweit gehen und die Prä-Integrationstabelle als 3D-Textur auf der Graphikkarte ablegen. Durch die Zuweisung von Texturkoordinaten, die den jeweiligen drei Integrationsparametern entsprechen, erledigt die Graphikhardware die komplette Darstellung eines Tetraeders. Das vorgestellte Verfahren ist also nicht nur per-Pixel exakt sondern auch wegen der Hardwarebeschleunigung entsprechend effizient.

Durch die zunehmende Flexibilität der letzten Graphikhardwaregeneration ist es weiterhin gelungen auch den Texturspeicherverbrauch, der mit der Prä-Integrations-Methode verbunden ist, zu reduzieren. Die hierfür verwendeten Techniken werden in Kapitel 10 beschrieben. Zum einen wird die drei-dimensionale Prä-Integrationstabelle, die sehr viel Texturspeicher belegen kann, durch Polynome approximiert. Schon Polynome vierter Ordnung erzielen eine hervorragende Genauigkeit. Dadurch kann die Tabelle von drei auf zwei Dimensionen reduziert werden. Die Rekonstruktion der Prä-Integrationstabelle erfolgt dann simultan aus den Polynomkoeffizienten im Pixel-Shader. Dies hat zusätzlich den Vorteil höherer interner Genauigkeit. Da die eigentliche Prä-Integration aber nach wie vor eine sehr hohe Anzahl numerischer Integrations-Schritte erfordert, treten bei einer Änderung der Transferfunktion hohe Wartezeiten auf. Auch dieses Problem lässt sich lösen, indem man die Graphikhardware geschickt einsetzt. Durch Verlage-

rung der Prä-Integration auf die Graphikhardware lassen sich Geschwindigkeitssteigerungen von bis zu 700% erreichen. Aus nahe liegenden Gründen wird diese Methode daher als hardwarebeschleunigte Prä-Integration bezeichnet.

In den Kapiteln 11 und 12 werden schließlich die beschriebenen Verfahren angewendet, um natürliche volumetrische Effekte darzustellen. Kapitel 11 widmet sich dem so genannten Bodennebel. Das ist eine Form von Nebel die durch die maximale Höhe der Nebelschicht über der Erdoberfläche beschrieben werden kann. Obwohl dies keine universelle Darstellungsform von Nebel ist, eignet sie sich nichtsdestotrotz hervorragend für Computerspiele, die Bodennebel als spielbestimmendes Element einsetzen.

Bodennebel wird wie folgt visualisiert: Aufbauend auf dem Dreiecksgitter, das der Terrain Renderer erzeugt hat, wird für jedes Basisdreieck ein Prisma generiert, das auf das Dreieck aufgesetzt wird. Die Höhe des Prismas ergibt sich aus der Höhe des Bodennebels an jener Stelle. Auf diese Weise ergibt die Menge der so generierten Prismen eine Nebelschicht. Jedes der Prismen wird in drei Tetraeder zerlegt, die wiederum mittels Zellprojektion gerendert werden. Da hierdurch schnell eine nicht zu vernachlässigende Anzahl von Tetraedern entsteht, kann man davon ausgehen, dass der Aufwand für die Darstellung des Bodennebels um einiges größer ist als der für die Darstellung des Terrains. Um diesen Nachteil auszugleichen, wählen wir ein einfaches optisches Modell. Für ein rein emissives Modell kann zur Darstellung der Prismen die PCP-Methode verwendet werden. Diese ist deutlich effizienter, hat aber den Nachteil der nicht-photorealistischen Darstellung des Nebels. Für Computerspiele ist dieser Nachteil aber sicherlich von zweitrangiger Bedeutung, da die Performanz hier Vorrang hat. Für die Zell-Projektion eines jeden Tetraeders sind eine Reihe von geometrischen Operationen notwendig, die in Ihrer Summe die Geschwindigkeit limitieren. Der PCP-Algorithmus hingegen berechnet die Projektion nicht direkt, sondern nutzt das vereinfachte optische Modell aus, um die Länge eines jeden Strahlsegments mit Hilfe der Graphikhardware zu berechnen. Dies geschieht in einem Multi-Pass-Verfahren, welches zuerst den Abstand der rückwärtigen Begrenzungsflächen eines Tetraeders zum Augpunkt bestimmt und im Bildschirmspeicher ablegt. Wiederholt man nun diese Prozedur für die Vorderseiten des Tetraeders und subtrahiert die beiden Abstandswerte voneinander, so erhält man die Länge eines jeden Strahlsegments. Die Strahlsegmentlänge wird nun mit dem Durchschnittswert der Eigenemission multipliziert, wodurch effektiv das Strahlintegral für das vereinfachte rein emissive optische Modell berechnet wird. Dadurch läuft der PCP-Algorithmus komplett auf der Graphikhardware.

In Kaptitel 12 schließlich wird die effiziente volumetrische Darstellung von Wolken behandelt. Im Prinzip beinhaltet dies auch die Darstellung von Bodennebel. Im Fall von Bodennebel ergeben sich aber aufgrund der Einfachheit der Problemstellung die beschriebenen Optimierungsansätze, weshalb Bodennebel expli-

zit behandelt wurde. Grundsätzlich können Wolken durch ihre optische Dichte, d.h. durch ein drei-dimensionales reguläres Skalarfeld definiert werden. Würde man nun jeden Voxel des regulären Gitters in Tetraeder zerlegen und diese mittels Zellprojektion zeichnen, so erhielte man schnell eine Menge an Tetraedern die sich in keinster Weise mehr interaktiv verarbeiten lässt. Man ist also bei einem Zellprojektionsansatz darauf angewiesen, den Datensatz entsprechend zu vereinfachen. Aufgrund der großen Ausdehnung der verwendeten Wolkendaten ist ein augpunktabhängiges Verfahren eine gute Wahl:

Analog dazu, wie der C-LOD Ansatz benutzt wird, um die Anzahl der Dreiecke eines Terrains zu verringern, kann dieser Ansatz ebenso die Anzahl der Tetraeder eines Volumens reduzieren. In Kapitel 12 werden daher die notwendigen Voraussetzungen und Algorithmen diskutiert, um den C-LOD Ansatz an die Visualisierung von drei-dimensionalen Skalarfeldern anzupassen. Obwohl der C-LOD Ansatz schon seit Jahren für die Landschaftsvisualisierung eingesetzt wird und dementsprechend weit verbreitet ist, wurde bisher interessanterweise eine analoge Formulierung für 3D Skalarfelder nicht vorgeschlagen. Nichtsdestotrotz zeigen unsere Ergebnisse, dass sich dieser Ansatz letztlich auch für die effiziente Visualisierung von Wolken eignet (siehe Abbildung 12.4).

Das Vorgehen ist hier wie folgt: Der reguläre Volumendatensatz wird als Octree repräsentiert. Für jeden Knoten des Octrees entscheidet eine Verfeinerungskriterium ob der Datensatz schon mit hinreichender Genauigkeit abgebildet wird. Ist der Fehler im Bildraum größer als eine vordefinierte Schranke, wird der betrachtete Knoten in acht kleinere Knoten zerlegt, die den Datensatz entsprechend genauer approximieren. Diese Prozedur wird solange rekursiv wiederholt bis keine Verfeinerung des Octrees mehr notwendig ist. Aufgrund des blickpunktabhängigen Verfeinerungskriteriums werden kleine und entfernte Details mit geringerer Priorität behandelt als solche die entsprechend nah sind. Dadurch ist eine optimale Darstellungsqualität bezogen auf die jeweilige Fehlerschranke gewährleistet.

Jeder Knoten des erzeugten Octrees wird nun in 5 Tetraeder zerlegt, die mittels Zellprojektion gezeichnet werden. Eine Beschleunigung mit Hilfe der PCP-Methode ist hier nicht möglich, da ein emissives Modell die Lichtverhältnisse innerhalb der Wolken zu stark vereinfachen würde. Bei augpunktabhängigen Verfahren hat man prinzipbedingt mit dem sogenannten “Popping Effect” zu kämpfen. Dies bedeutet, dass ein entferntes Objekt, das zunächst zu klein ist, um dargestellt zu werden, ab einer bestimmten Distanz plötzlich auftaucht. Da das menschliche Auge auf temporäre Änderungen besonders empfindlich reagiert, muss dieser Effekt auf jeden Fall unterdrückt werden. Könnte man den Fehler im Bildraum auf unter einen Pixel drücken, so wäre der Effekt nicht wahrnehmbar. Die Geschwindigkeit aktueller Graphikhardware ist jedoch für dieses Anwendungsgebiet noch nicht schnell genug. Man erreicht daher bestenfalls eine Fehlerschranke

von etwa 10 Bildpunkten. Ein Ansatz zur Lösung des Problems ist, die einzelnen Detailstufen langsam ineinander übergehen zu lassen, d.h. diese zu interpolieren. Dadurch erscheint ein kleines Detail nicht mehr plötzlich sondern vielmehr langsam und fließend. Da für die Interpolation der Detailstufen eine Vielzahl an zusätzlichen Fliesskommaoperationen notwendig sind, ist der in der Dissertation vorgestellte Algorithmus speziell daraufhin optimiert.

Abschließend werden die in der Dissertation entwickelten Verfahren den bereits bekannten Algorithmen gegenübergestellt. Es werden Entscheidungskriterien vorgestellt, welche die Auswahl der jeweils passenden Methode für ein spezielles Anwendungsgebiet erleichtern. Insgesamt betrachtet wurden in der Dissertation zwei fundamental neue Volumenvisualisierungstechniken vorgestellt: Zum einen die **PCP-Methode** und zum anderen die **prä-integrierte Zellprojektion**. Diese Methoden erweitern das Einsatzspektrum von unstrukturierten Volumenvisualisierungsmethoden sowohl in Hinsicht auf Bildqualität als auch in Hinsicht auf Darstellungsgeschwindigkeit. Das beste Beispiel dafür sind die neuen Algorithmen zur Darstellung von Bodennebel und volumetrischen Wolken.