Advanced Techniques for the Rendering and Visualization of Volumetric Seismic Data

M.Sc. Thesis
by Martin Panknin

Supervision:

Prof. Dr. Christian Geiger
Prof. Dr. Sina Mostafawy

External Supervision:
David d'Angelo M.Sc.

University of Applied Sciences Düsseldorf
Department of Media

Submitted on 17.04.2012

Statement of Originality

Hereby I declare that I wrote this master thesis without any assistance from third parties and without sources other than those indicated in the thesis itself.

Martin Panknin

Contents

List of Figures

List of Tables

Acknowledgement

At first, I would like to thank Prof. Dr. Christian Geiger and Prof. Dr. Sina Mostafawy for supervising this thesis.

Moreover I would like to thank all my colleagues at the Fraunhofer IAIS. I especially thank David d'Angelo, Dr. Manfred Boden, Dr. Stefan Rilling and Thorsten Holtkämper for their professional advise and for giving me the opportunity to write this thesis.

I thank all the members of the VRGeo consortium for their insightful comments, in particular Nicola Lisi and Roberto Lotti from ENI, Egil Tjaland from NTNU as well as Mark Dobin from ExxonMobil.

Further I am grateful to Annika Möser and Thomas Reufer for granting me the necessary open space during the last weeks of writing this thesis.

I would like to thank my entire family for their enduring support over the years.

Last but not least I would like to thank Sandra, without her help in all other matters, completing this thesis would not have been possible.

Chapter 1

Introduction

This chapter contains an overview of the contents of this work. An explanation is given on how the idea for this thesis originated followed by a brief description of the contents of each chapter. The thesis objectives will also be defined.

1.1 Motivation

An important part of today's search for hydrocarbon reservoirs such as oil and gas is the use of seismic methods which measure changes in acoustic impedance to explore the interior of the earth. Similar to medical imaging techniques such as MRI or CT, seismic methods generate image slices (survey lines) through the subsurface geology. By placing many of these 2D survey lines within close distance to each other a more detailed three-dimensional picture can be build. This technique is called a *3D survey* [Ric05, p. 123].

The visualization and interpretation of the resulting information is a challenging task due to the huge amount of generated data, which can easily reach sizes of multiple gigabytes [Mur02]. Moreover, seismic data contains a high presence of noise, which makes it even more difficult to visualize. Interpretation often includes time-consuming manual tasks to identify subsurface structures that may hint at potential hydrocarbon deposits.

An often used method to support the interpretation of volumetric seismic data is volume rendering. This thesis mainly focuses on techniques to improve the classification and visualization of volumetric seismic data by means of volume rendering.

More specifically it will be investigated if the so-called *occlusion spectrum*, which is a classification technique from the medical domain, can be used in the context of seismic

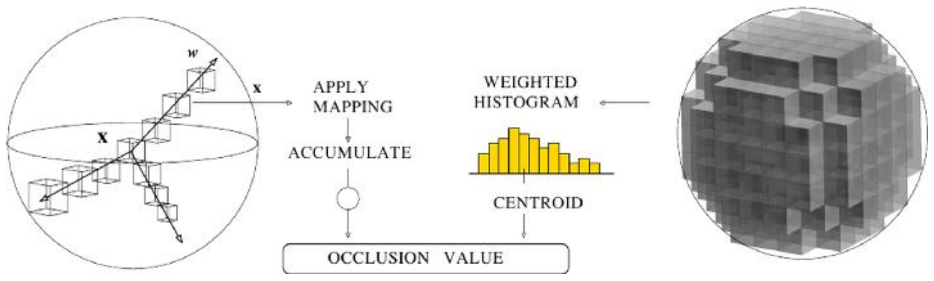


Figure 1.1: Occlusion is the weighted average of the visibility of a point along all direction in a spherical neighborhood [CK09, p. 3].

volume rendering.

1.2 The Occlusion Spectrum

The occlusion spectrum is a technique for the classification of volumetric datasets based on *ambient occlusion* of voxels [CK09]. The term ambient occlusion at this point is borrowed from the rendering domain, where similar algorithms are used to approximate global illumination effects due to the occlusion of objects. In this context however it is used for the classification of volumetric datasets.

The occlusion spectrum is calculated using techniques similar to the generation of classical ambient occlusion. In a preprocessing step a weighted average of a spherical neighborhood for each voxel is built. This is done by sampling all voxels that are located within a given area. The intensities of the surrounding voxels are weighted according to a *visibility mapping* and then accumulated. This accumulated value encodes the weighted average visibility of a point along all directions in a sphere. During rendering, occlusion is used together with the original signal intensity to generate two-dimensional transfer functions (see chapter 3).

Contrary to other classification methods such as gradient magnitude, the occlusion spectrum highlights structures inside the data instead of material boundaries [CK09]. The original paper describes this technique mostly by means of medical datasets such as MRI or CT data. The idea of this thesis is to adapt this technique in order to classify and visualize volumetric seismic data sets from the oil and gas domain. The goal of this adaption is the better identification of structures such as salt domes or channel systems, which are an integral part of the interpretation of 3D surveys.

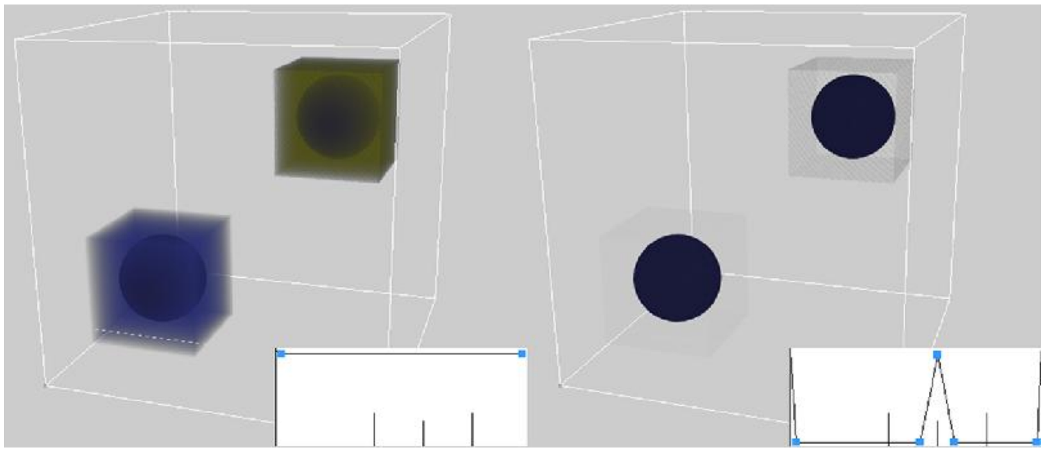


Figure 1.2: Two spheres with same intensity surrounded by two cubes (left). Cubes can be hidden by mapping intensity values to opacity (right).

Basic Example

The concept of occlusion-based classification can be explained with the following example. The synthetic dataset in Figure 1.2 consists of two spheres that share the same intensity. One is surrounded by a cube with lower intensity, the other with high intensity. By mapping the intensity of voxels to transparency the spheres can be separated from the cubes. However it is not possible to further differentiate between the two spheres, as they have the exact same intensity. It is only possible to completely hide or display both spheres.

By further considering the direct neighborhood of the spheres they can be separated based on the amount of occlusion. The sphere that is surrounded by the cube with the high intensity lies in the upper occlusion range whereas the neighborhood of the other sphere shows a lower occlusion distribution (Figure 1.3). The classification is based on the *distribution of intensities* around volumetric objects.

Figure 1.4 shows a more advanced example where an MRI of a human head is classified in terms of occlusion. Depending on the selection inside the occlusion spectrum, different anatomical structures can be displayed separately even though they share the same intensity.

Visibility Mapping

In the above example occlusion is generated in a linear way. This means that low intensities correspond to low occlusion whereas high intensities correspond to high occlusion values. This characteristic can be expressed graphically with a simple linear function (Figure 1.5,

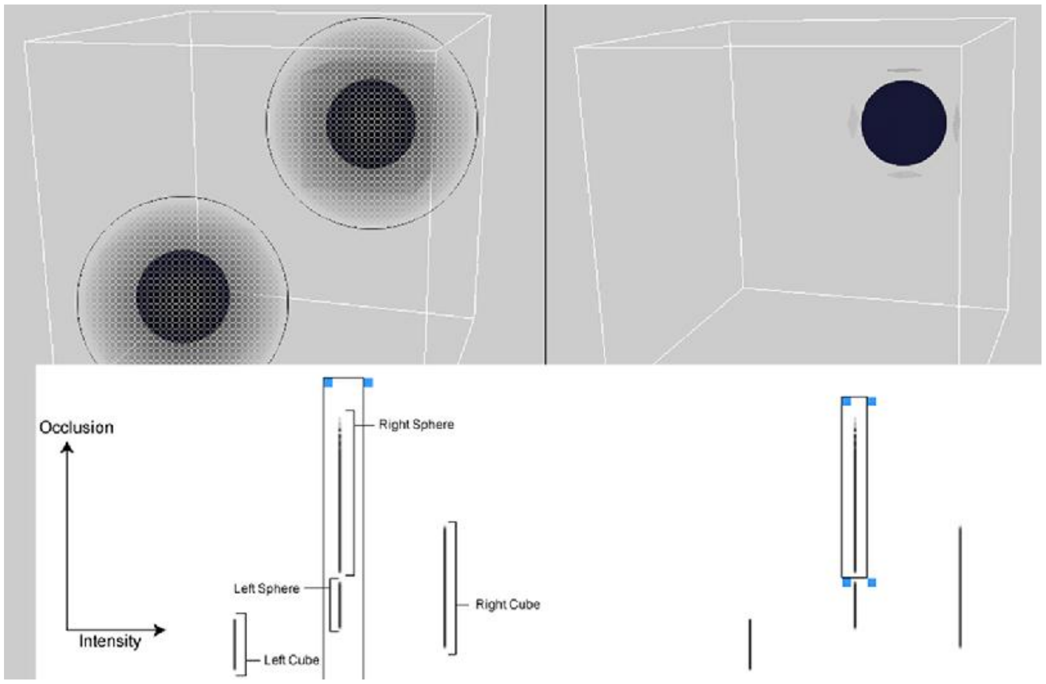


Figure 1.3: Based on the distribution of intensities around the spheres, they can be separated from each other. The blue selection points define the visible portion of the data set.

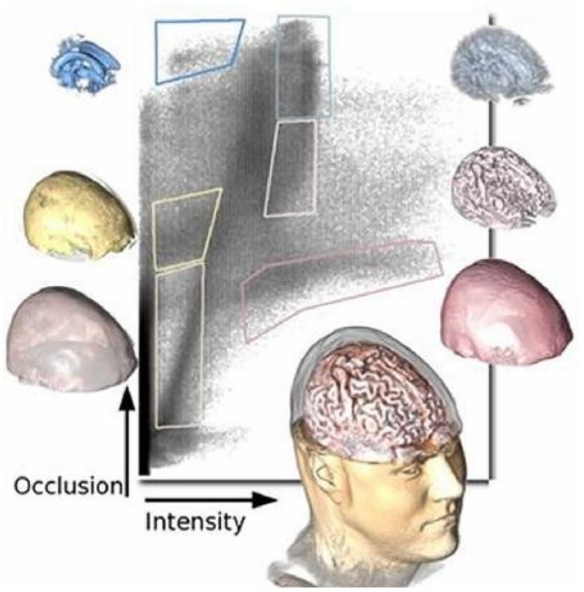


Figure 1.4: Occlusion-based classification of an MRI of a human head [CK09, p. 2].

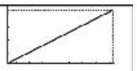
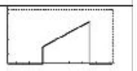
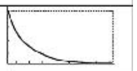
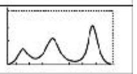
Linear Ramp	$M(\mathbf{x}) = S(\mathbf{x})$	
Truncated Linear Ramp	$M(\mathbf{x}) = \begin{cases} S(\mathbf{x}) & \tau_0 < S(\mathbf{x}) < \tau_1 \\ 0 & \text{otherwise} \end{cases}$	
Distance weighted	$M(\mathbf{x}) = S(\mathbf{x})e^{-\lVert\mathbf{x}-\mathbf{x_c}\rVert^2}$	
Opacity weighted	$M(\mathbf{x}) = \alpha(\mathbf{x})$	

Figure 1.5: Different visibility mappings used during occlusion generation [CK09, p. 3].

top). The way different intensities contribute to the occlusion is controlled with a *visibility mapping*. Figure 1.5 shows different possible mapping functions. The mapping function has a great impact on the final occlusion spectrum, because it can be used to emphasize or completely hide intensity ranges during occlusion generation. Consider the truncated mapping. During occlusion preprocessing, each voxel that is located in either the minimum or maximum intensity ranges will not contribute to the occlusion. Choosing a suitable visibility function is therefore of great importance for the final quality of the classification. It can be used to vertically spread certain intensity ranges, which is important in order to maximize the likelihood of structure separation [CK09]. Therefore a good occlusion spectrum contains a lot of variance in the vertical dimension.

1.3 Objectives of this Thesis

The goal of this work will be to investigate if and how the occlusion spectrum can be applied to volumetric datasets from the geoscientific domain. Therefore one part of this thesis will be the conception and the prototypic implementation of a system which can be used to classify volumetric seismic data using the occlusion spectrum. To validate the results on the one hand but also to enable the processing of bigger volumes on the other it is necessary to develop a fast method for the preprocessing calculations. Thus, this work includes the development of a new algorithm for the generation of the occlusion values from the original intensity volume. To evaluate the results, synthetic datasets will be constructed that try to simulate the data distribution of seismic data.

1.4 Structure of this Thesis

A basic introduction into petroleum geology is given in chapter 2. In addition there will be an overview of certain geophysical survey techniques including seismic reflection surveying followed by a short description of how seismic measurements are transformed into renderable volumetric datasets. Chapter 3 deals with volume rendering and explains basic principles and concepts of rendering discrete scalar fields such as volume datasets. One important part of this work is the development of a novel algorithm for precomputing occlusion information from a volume dataset. Since this algorithm is implemented in a parallel fashion for processing on modern GPUs, chapter 4 deals with the topic of GPU computing. Besides a general introduction there will be a more detailed description of Nvidia's CUDA programming architecture. A description of the new sampling algorithm is given in chapter 5 including detailed comparisons of execution times and complexity between the original CPU-based version. Chapter 6 deals with an adaptive approach for selecting an optimal visibility mapping for occlusion computation. In the following chapter, the implementation of the demo application will be explained in some detail. The thesis ends with a discussion of results in chapter 8 and conclusion and future work in chapter 9.

1.5 Fraunhofer IAIS and the VRGeo Consortium

On behalf of the VRGeo Consortium, which represents the international oil and gas industry, their suppliers of software applications, and their providers of related technology, the Fraunhofer Institute for Intelligent Analysis and Information Systems (Fraunhofer IAIS) is engaged in the development and evaluation of interactive hardware and software technologies for visualization systems in the geoscientific domain [VRG12]. In this regard the results of this thesis will be presented to the consortium members at the VRGeo meeting which takes place twice a year to discuss latest research results.

Chapter 2

Geological Background

In order to locate natural oil or mineral deposits the goal of many geophysical exploration efforts is to identify contrasts to the general background of data [Ric05, p. 19]. Based on this informations conclusions can be drawn regarding potential hydrocarbon reservoirs. This chapter will give a basic introduction on how natural hydrocarbons deposits are formed and in what kind of geological structures they can be found. Furthermore an introduction is given regarding the geophysical methods that are applied when performing petroleum exploration. Due to its importance in hydrocarbon exploration, seismic survey techniques will be described in more detail and certain theoretical aspects in respect to seismic waves and seismic reflection techniques will also be discussed.

2.1 Formation of Petroleum Deposits

Natural oil and gas fields develop over long periods of time, slowly transforming organic matter through chemical conversion processes into hydrocarbons. Dead marine organisms such as algae accumulate over hundreds of thousands of years on the sea floor, building sediment stratas with a high proportion of organic material. Over time more and more sediments are deposited on top of the organic layer, creating higher pressures and temperatures. Under these effects of pressure and heat the hydrocarbons that are contained in the biomass distill into crude oil or natural gas. Rocks that contain accumulated amounts of these distilled hydrocarbons are called *source rocks*.

If sediment layers above a source rock are porous and permeable the interstitial oil may flow out of the rock and travel through more porous limestone or sandstone until it

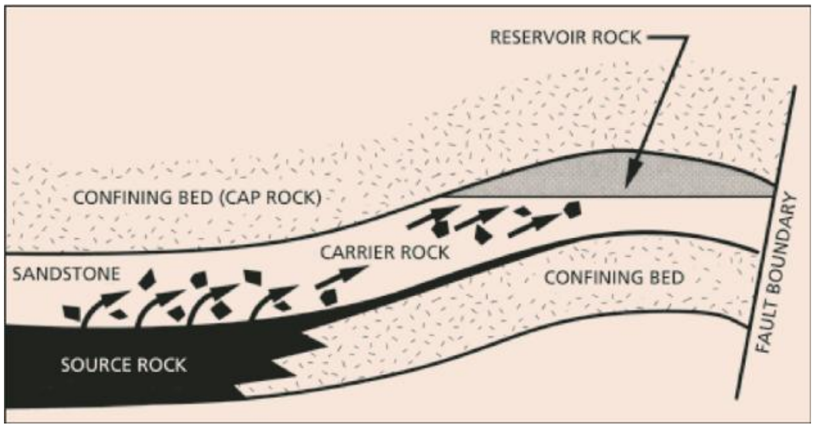


Figure 2.1: Oil migration from source into reservoir rock [Ric05, p. 10].

reaches a *reservoir* or is blocked by an impermeable layer. This process of movement is called *petroleum migration* and is caused by hydrology, fluid pressures and water movement. Once the movement of oil or gas has come to an end because it reached an impermeable sediment layer, it is said to be trapped and the geological structure that causes the end of the hydrocarbon migration is called *petroleum trap* [Ric05]. There are four main groups of geological structure in which hydrocarbons can be trapped: *anticlines*, *salt domes*, *faults* and *unconformities* (Figure 2.2). However for deposits to develop several conditions must be fulfilled. A source rock must be present and located within a suitable geometrical and historical relations to a reservoir or a trap. Moreover there must be a migration pathway that actually allows the petroleum to travel into the reservoir. The fact that source rock and petroleum trap can be separated by very long distances makes the detection of reservoirs even more complicated. It can be seen that the accurate detection of such geological features is a complicated task since no direct picture of the subsurface can be taken. Information on the inner depths of the earth must be derived from results of different geophysical measurements.

2.2 Survey Techniques

The most unambiguous tool for locating hydrocarbon reservoirs is to drill. However the process of drilling requires an enormous financial investment and is extremely labor intensive. Moreover, drilling blindly into the subsurface carries great risks for nature and possibly also the human population and therefore it is crucial to obtain as much information as possible

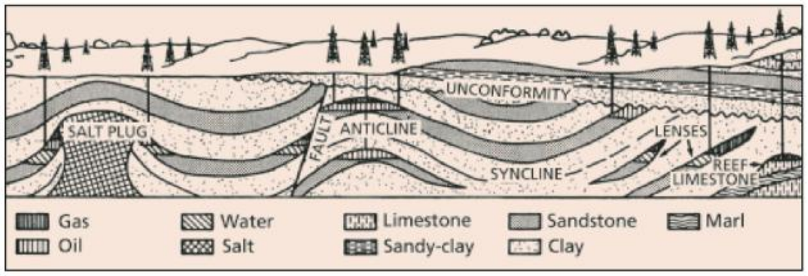


Figure 2.2: Different types of hydrocarbon traps [Ric05, p. 17].

on the target area to be able to properly prepare a drilling operation. Geophysical *surveys* cover a wide range of techniques and processes that aim at reducing the potential risks and maximize the profit outlook of drilling operations. Survey techniques can be classified as either *active* or *passive* depending on whether they make use of fields that occur naturally or whether they take measurements of artificially generated energy. Examples of natural fields are the gravitational, magnetic, electrical or electromagnetic fields of the earth [KBH02].

Artificially generated sources can be the generation of local electrical or electromagnetic fields or most importantly in the context of hydrocarbon exploration seismic waves that are used to explore the inner depths of the earth. Geophysical methods are generally used in a complementary sense, since none of the techniques delivers unambiguous evidence of the subsurface. For example the beginning of a survey might be done by airborne-based gravimetric and magnetic measurements. Based on the findings of these measurements certain areas can be identified where more detailed seismic surveys are carried out subsequently.

2.2.1 Non-Seismic Survey Techniques

There are various geophysical techniques that are applied before drilling and some of them will be described in this section. Seismic techniques are the most widely used type of examination method in the area of petroleum exploration and will be described in more detail at the end of this section.

Gravimetric Surveys

Variations in subsurface rock density may influence the strength of gravity above it. Measurements of these gravity variations can be used to draw conclusions regarding geological

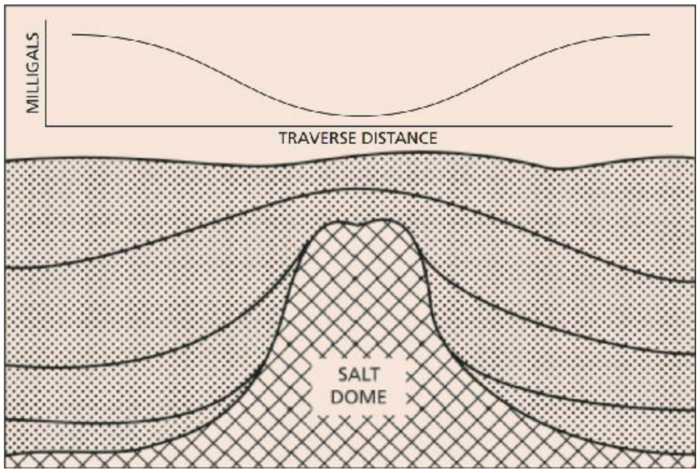


Figure 2.3: Gravity anomalies relate to geological structure. Low values are caused by light rocks [Ric05, p. 20].

structures. The main targets of this survey type are mostly large-scale subsurface structures, as smaller structures would not disturb the earth's gravitational field enough to be detectable at the surface level. Large salt domes for example have a low density compared to the overlying strata, which causes them to rise upwards due to buoyancy forces. This difference in density is detectable since the gravity above the dome is lower compared to the gravity above more dense rock structures (Figure 2.3). Measurements can hence be used to infer the position and the size of the dome. Gravimetric surveys are useful for gaining an initial overview of an area of interest [Ric05, p. 19].

Magnetic Surveys

Since rocks contain a small amount of ferromagnetic minerals, they all have a weak magnetism that influences their ambient magnetic field. Magnetic surveys respond to these local variations in the magnetic field. Magnetic measurements respond to changes in rock magnetization. Sedimentary basins are generally very low or non-magnetic, which makes them clearly separable from surrounding rocks that tend to have a stronger magnetism. In the context of magnetic exploration this is an important property and can used to outline the shape of a reservoir [Ric05, p. 23]. Similar to gravimetric techniques, magnetic observations can serve as a valuable primary exploration tool.

Electrical Surveys

Electrical surveys can be either active or passive. Passive methods measure electrical fields that may be present naturally, while others introduce artificially generated currents. The *resistivity method* for example can be used to measure potential differences of artificially generated electric currents. These currents are introduced into the ground and the potential differences are measured at the surface [KBH02, p. 183]. The results of these measurements deliver information on the electrical properties of the subsurface. One advantage of electrical surveys is the ability to investigate the more shallow subsurface, which might not be well suited to seismic methods for example, since these can only be used to retrieve images of structures below approximately 50 feet. Further information regarding electrical surveys can be found in [KBH02].

Electromagnetic (EM) Surveys

One application case for EM surveys is Sea Bed Logging. SBL is performed offshore by using a mobile horizontal electric dipole (HED) source transmitting a low frequency electromagnetic signal which is picked up by multiple arrays of electric field receivers that are placed on the sea floor. SBL can be used to distinguish whether a reservoir is filled with water or hydrocarbons. This is due to the hydrocarbons having a higher resistivity compared with shale or reservoirs filled with water [TE02].

2.2.2 Seismic Survey Techniques

The basic principle of seismic measurements is very similar to the one of ultrasound imaging, which produces images of the interior of a human body using very high frequency sonic pulses. In contrast to the very short wavelength of ultrasound, seismic methods use far lower frequencies resulting in much longer wavelengths. So one major difference between the two techniques is the distance, which the sound waves can travel into the source object. High-frequency ultrasound penetrates a human body in the centimeter range, whereas the low-frequency seismic pulses can reach depths of several thousand meters inside the earth. To perform a seismic measurement it is necessary to generate an artificial low-frequency sound wave at the surface level. This wave will travel down into the earth and will be partially refracted and reflected at the interfaces between different rock layers. Each time a wave hits an interface it is split and partitioned into two reflected and two refracted waves.

In the context of reflection seismology this is called a *seismic event*. The reflected energy travels back to the surface and is registered by multiple arrays of geophones. During this process two important properties can be measured.

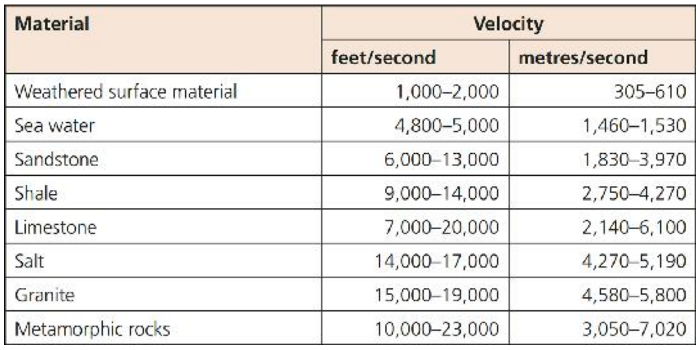
Two Way Travel Time

The first is the amount of time it takes for the wave to travel down to the reflector and back up to the receiver. This temporal information is related to the *depth* of a reflector and is known as two way travel time (TWTT). However only reflections that are below 50 feet can be detected clearly because arrival times of reflectors above that depth are nearly the same as the much stronger direct ground reflection. Reflections of deeper interfaces will arrive at the geophones later and after the peak amplitude of the ground reflection has passed, which makes them easier to detect [Env03].

Reflection Strength

The second value is the strength of the reflected signal, which is an indicator of the properties of rock change at the reflecting interface. The amount of energy that is reflected at an interface depends on the contrast in *impedance* between two overlying layers [Low07]. Acoustic impedance is defined as the product of density and seismic velocity in a certain rock layer and is measured in distance per second. Seismic velocity is the propagation velocity of a seismic wave for a given medium and depends on multiple factors such as porosity, lithology, interstitial fluids or depth. For example since rock density tends to increase with depth, the speed of seismic waves also becomes faster the deeper a wave has penetrated into the earth. Table 2.4 contains average wave velocities for materials that are commonly contained in the interior of the earth.

Seismic measurements do not directly locate the position of hydrocarbon reservoirs but instead deliver important structural information that can be used to infer the presence or absence of oil and gas. This is due to certain properties such as source rock maturity or migration pathways not being detectable in seismic data [Kam10]. Seismic techniques are therefore sometimes described as *indirect* exploration techniques. Nevertheless nowadays the seismic method is by far the most widely used technique in the field of geophysical exploration.



Material	Velocity	
	feet/second	metres/second
Weathered surface material	1,000–2,000	305–610
Sea water	4,800–5,000	1,460–1,530
Sandstone	6,000–13,000	1,830–3,970
Shale	9,000–14,000	2,750–4,270
Limestone	7,000–20,000	2,140–6,100
Salt	14,000–17,000	4,270–5,190
Granite	15,000–19,000	4,580–5,800
Metamorphic rocks	10,000–23,000	3,050–7,020

Figure 2.4: Average seismic velocities for common materials [Ric05, p. 32].

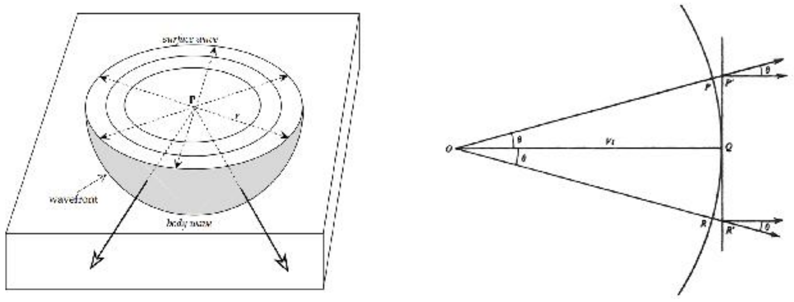


Figure 2.5: Propagation of a seismic wave from a point source P near the surface of a uniform medium (left) [Low07]. A spherical wavefront can be approximated by a planar wavefront (right) [RL95].

2.3 Seismic Data

2.3.1 Seismic Waves

Reflection seismology uses artificially generated seismic waves to explore the inner depths of the earth. Since these seismic waves are ordinary sound waves they spread out in all directions when traveling through a medium. The speed of propagation depends on the density of a material. The points of a wave that have traveled for the same amount of time are called *wavefront*. In a uniform medium the wavefront of a seismic wave has a spherical shape with all points of the wave being equidistant from the source (Figure 2.5). Points on a wavefront share the amount of time they have traveled but not necessarily the same amount of distance, as the propagation speed may vary due to different rock densities within the subsurface. Seismic waves can be divided into two main groups, *body waves* and *surface waves*.

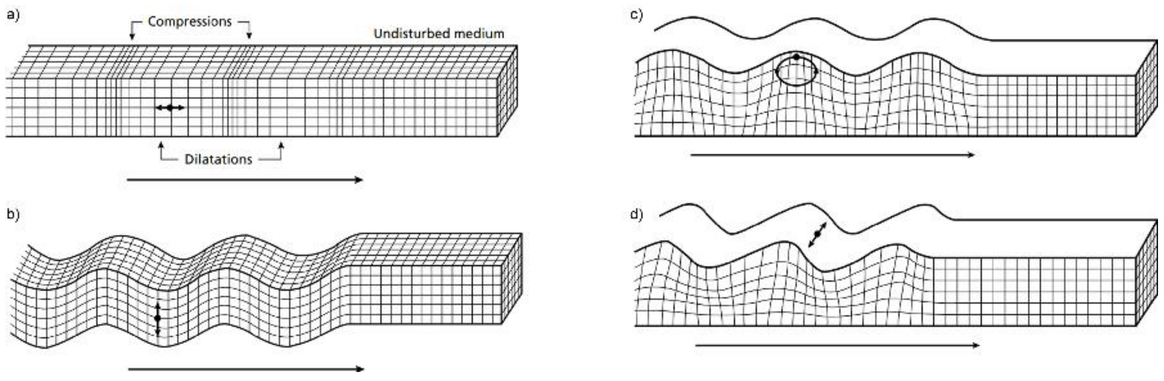


Figure 2.6: Displacement patterns for different wave types. Compressional waves (a), shear waves (b), Rayleigh waves (c) and love waves (d) [KBH02].

Body Waves

Body waves can occur either as *compressional waves* (longitudinal, primary or P-wave) or as *shear waves* (transversal, secondary or S-wave). Compressional waves move through a medium by axial deformation of the medium in the direction of wave propagation. This deformation consists of consecutive compressions and dilatations of the transmission medium. Shear waves propagate by a shear displacement that is perpendicular to the direction of the wave. Because compressional waves will always travel faster than shear waves in the same medium they arrive at the receiver first and are hence also known as primary waves [Low07]. Unlike compressional waves that can propagate in any kind of medium (solids, liquids and gases), shear waves can only propagate through solid materials [KBH02]. This characteristic leads to the assumption that the outer core of the earth is liquid since no shear waves can travel through it. Figure 2.6 illustrates the displacement patterns for different types of seismic waves.

Surface Waves

In contrast to body waves that spread out into the subject medium, surface waves travel in parallel along the boundary of a solid. In a geophysical context this means the crust of the earth. The name comes from the circumstance, that surface waves are tied to the surface and attenuate with distance from the surface. This means that amplitude and particle motion decrease rapidly with increasing depth. However since surface waves only propagate in two dimensions, they lose less energy and therefore decay more slowly compared to body waves, which propagate in three dimensions. There are two types of surface waves that are of importance in the context of reflection seismology, *Rayleigh waves* and *Love waves* (Figure 2.6). The former propagate similar to waves on the surface of an ocean. The transmission medium is moved up and down and side to side perpendicular to the direction of the wave. This includes longitudinal as well as transversal particle motions. The latter are the fastest moving surface waves and propagate by horizontal shifting of the earth and are sometimes described as polarized shear waves. Surface waves travel generally with a slower velocity compared to body waves [KBH02].

Seismic Raytracing

Determining the exact propagation paths (thus the wavefront) of a seismic wave through a heterogeneous medium such as subsurface geology is an extremely difficult task. In order to simplify the calculations the following mathematical simplifications are applied. One observation that can be made when looking at the spreading of a spherical wave is that the curvature of the wavefront decreases with distance from the source. At great distances the radius becomes very large and an infinitesimal small section of the wavefront approaches a *plane wave*. This is illustrated in Figure 2.5. Since these are simpler to visualize and to express mathematically seismic waves are generally treated as plane waves [RL95].

The second simplification is that the direction of wave propagation can be described by rays which are perpendicular to the wavefront and point away from the source of the wave. These imaginary rays are called seismic *ray paths* and the process of tracking the propagation of these rays through a given medium is called seismic *ray tracing*. The method of seismic ray tracing is used to generate structural subsurface models based on calculated and observed travel times of both refracted and reflected seismic disturbances [KBH02]. Since the same concepts that describe reflection and refraction of light rays are also valid for seismic energy, studying the ray paths can deliver evidence on the position or shape of certain subsurface structures.

2.3.2 Seismic Volumes

Seismic Traces

Seismic amplitudes are generally displayed in the form of *seismic traces*. A single trace is a graph of signal amplitude against TWTT and represents originally the recorded signal from

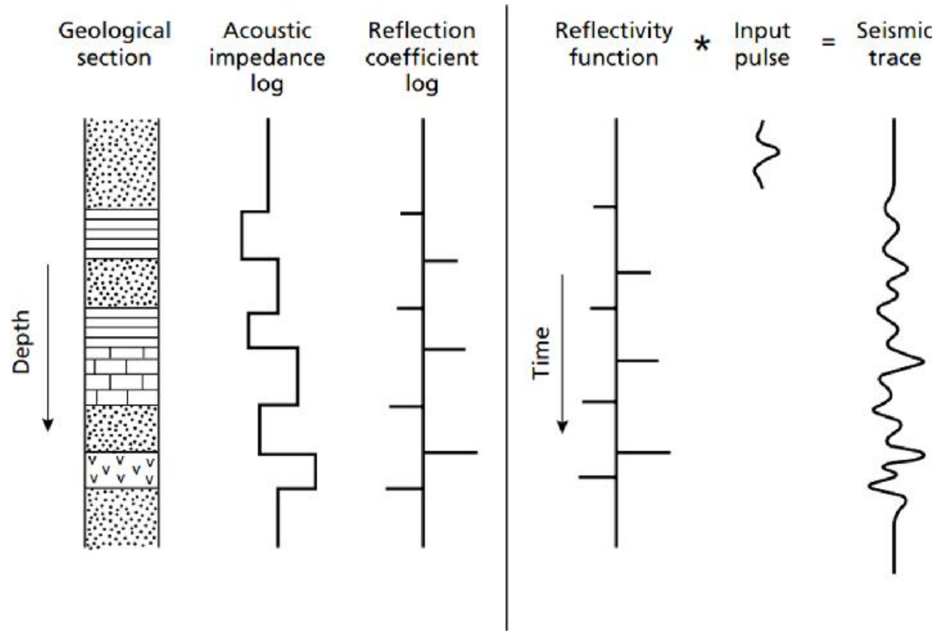


Figure 2.7: The convolutional model of the reflection seismic trace, showing the trace as the convolved output of a reflectivity function with an input pulse, and the relationship of the reflectivity function to the physical properties of the geological layers [KBH02].

a single seismograph. The depth of a reflection can be derived from the travel time and the strength of amplitude peaks relates to the contrast in acoustic impedance of two adjacent layers (Figure 2.7). The resulting trace can be described as the convolution of an input pulse with a *reflectivity function* that itself is based on the acoustic impedance distribution [KBH02].

Seismic Lines and Volumes

By taking a number of these measurements at regular distances to each other (thus placing multiple traces in a row), a two-dimensional image can be constructed that represents a cross section through the earth. This image is called a *seismic line.* By stacking multiple lines a three-dimensional image (3D survey) of the subsurface can be built. This image represents a discrete scalar field with the scalar values relating to seismic amplitudes and grid points relating to the spatial coordinate of the respective seismic events. Since seismic volumes can be represented as discrete scalar fields, they can be displayed in terms of volume

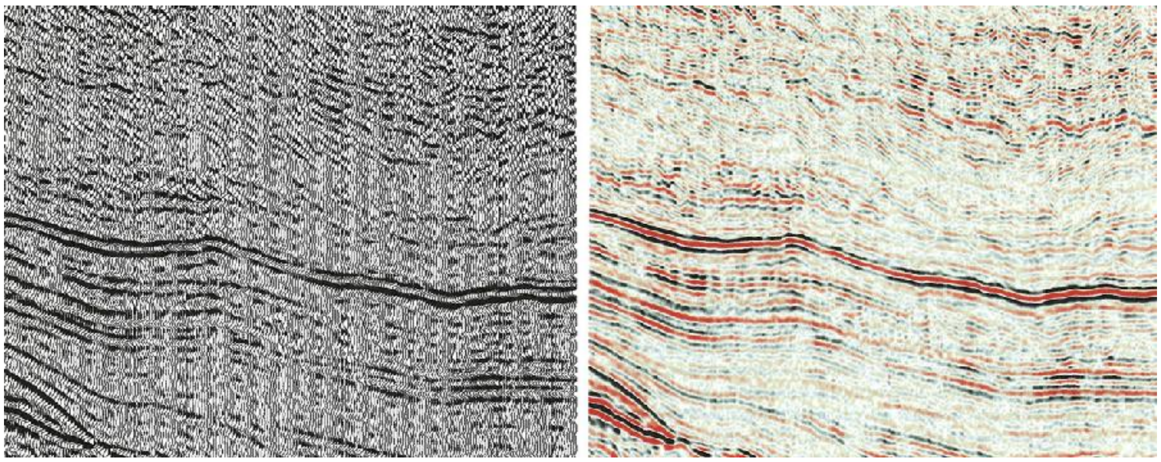


Figure 2.8: Seismic lines consist of multiple seismic traces placed in a row. Seismic lines can be either displayed as wiggle plot (left), or by applying color tables that map amplitudes to optical properties such as color and opacity (right) [Bro04].

rendering (see chapter 3). Such a discrete seismic volume can be described as,

$$X(t, x, y) \epsilon \mathbb{R}, \tag{2.1}$$

where x and y are spatial coordinates and t is the travel time TWTT [PMM03]. A single seismic trace is therefore described as the one-dimensional time-dependent function,

$$X(t, x_0, y_0). \tag{2.2}$$

The data samples in a seismic volume are normally distributed and a corresponding histogram will generally have the shape of a Gaussian bell. The minimum and maximum amplitude values (peaks and troughs) are caused by velocity anomalies due to the presence of fluids such as hydrocarbon accumulations. Amplitude values that are located around the zero point are caused by noise during data acquisition. The moderate amplitude intervals contain information on geological structures. Velocity anomalies can be identified and classified easily, by hiding all moderate and low amplitude ranges. Interpreters however spend most of their time analyzing the moderate amplitude ranges, where the structural information is contained [LBF05].

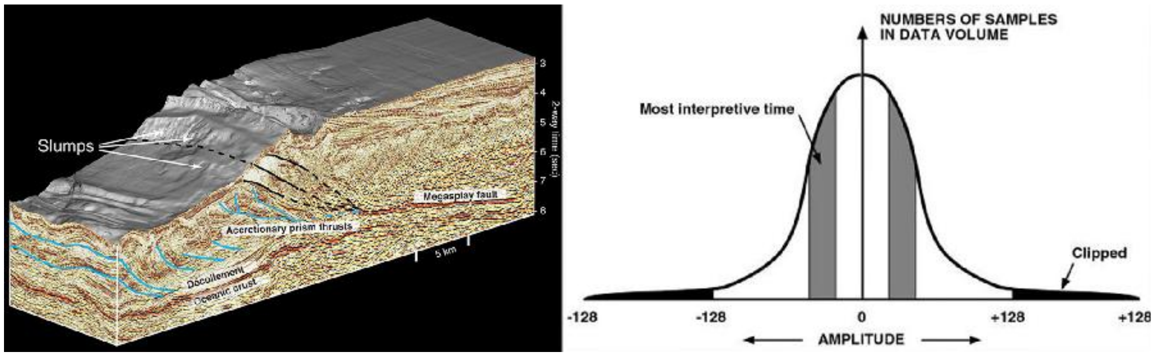


Figure 2.9: Seismic volumes are built by stacking multiple seismic lines (left) [G.F07]. Typical statistical data distribution in a seismic dataset (right) [Bro04].

2.3.3 Seismic Attributes

Seismic attributes are all the information obtained from seismic data, either by direct measurements or by logical or experience based-reasoning [M. 01]. Information relating to amplitude or position of the seismic waveform can be used to provide evidence on structural, stratigraphic or lithological parameters to an interpreter. Looking at seismic attributes may reveal certain features in the data that may not be apparent otherwise. In the context of hydrocarbon exploration this can be especially hydrocarbon accumulations [...].

Following the above definition, occlusion calculated for a seismic data set can also be described as a seismic attribute since it is essentially derived from seismic data. Seismic attributes can be classified in many different ways depending on the method of generation or based on domain characteristics. A complete discussion of seismic attributes is beyond the scope of this thesis but can be found in [M. 01]. As part of this work *sweetness* [Bru08] will be introduced as an example of a complex seismic attribute.

Complex Seismic Attributes

Complex attributes are all attributes that can be derived from the complex seismic trace which is calculated by a 90-degree phase shift of the recorded seismic trace (Figure 2.10), which is commonly done by a Hilbert transformation [Ame99a]. This leads to a complex function with the real part being the original seismic trace and the imaginary part being the phase-shifted trace. Plotting this function in the direction of reflection time leads to a helix-like structure. A projection onto the real plane gives the actual seismic trace and a projection onto the imaginary plane leads to the quadrature trace. This complex trace

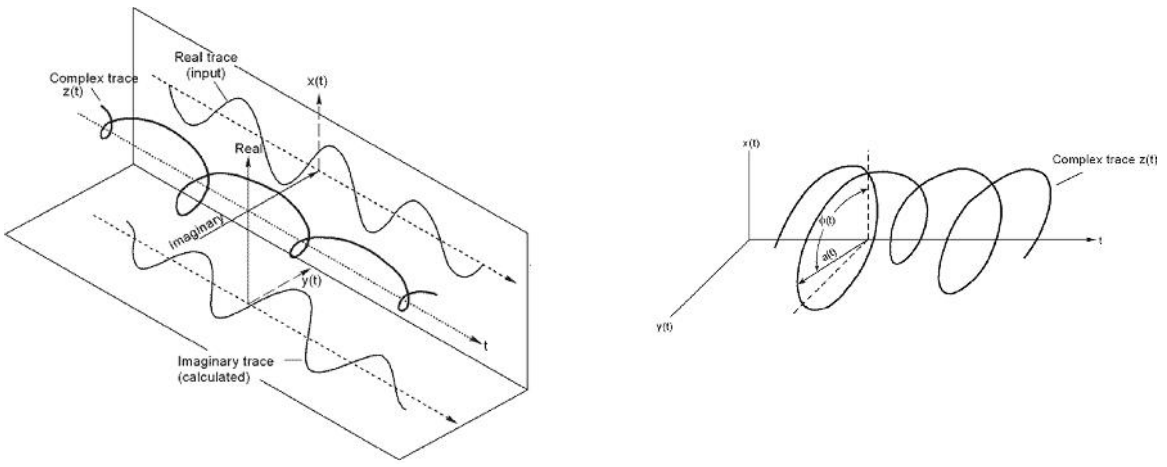


Figure 2.10: The complex seismic trace is generated by a 90 degree phase shift of the real seismic trace resulting in a helix like structure (left). The three key attributes can be calculated from a complex seismic trace (right) [Ame99b].

function is the basis for a whole range of seismic attributes which are generally referred to as *complex seismic attributes.* There are three key complex attributes - *instantaneous amplitude* $a(t)$, *instantaneous phase* $\phi(t)$ and *instantaneous frequency* $\omega(t)$. Instantaneous amplitude (also known as reflection strength or energy envelope) is defined as,

$$a(t) = \sqrt{x^2(t) + y^2(t)} \tag{2.3}$$

and instantaneous phase as,

$$\phi(t) = tan^{-1}(\frac{y(t)}{x(t)}) \tag{2.4}$$

and instantaneous frequency as,

$$\omega(t) = \frac{\partial}{\partial t}[\phi(t)]. \tag{2.5}$$

Seismic Sweetness

Seismic Sweetness [Bru08] is commonly used for identifying sands and sandstones. It combines instantaneous amplitude and instantaneous frequency and is calculated as,

$$s(t) = \frac{a(t)}{\sqrt{\omega(t)}}, \tag{2.6}$$

where $a(t)$ is the instantaneous amplitude and $\omega(t)$ is the instantaneous frequency. Both are complex seismic attributes and are derived from the complex seismic trace. Sweetness can be used to highlight major impedance contrasts that are good indicators of oil and gas deposits. The attribute was derived based on the observation that the presence of fluids generally increases the amplitude strength and also decreases the frequency content. Sweetness which combines the two attributes therefore gives a better image quality in these regions. It is also very suitable for detecting subsurface channel systems. In combination with other attributes such as *semblance* it helps to identify channel boundaries and get an impression of the interior of the channel. A high sweetness for example is generated by sands that contain hydrocarbons whereas lower values are caused by shale [Rus10].

Chapter 3

Volume Rendering

Volume rendering describes a range of techniques for displaying discrete scalar data fields such as volumetric data sets. These are generated in various areas including medical imaging, technical simulations, gaming and measurements from the geo scientific domain. There are different ways for acquiring the data, such as computer tomography, magnetic resonance imaging, ultrasound or seismic techniques. In these domains volume rendering is nowadays an essential tool for analyzing three-dimensional datasets. In addition to the field of scientific visualization, volume rendering also plays an important role in the generation of "'special effects"'. For example certain structures that do not contain an implicit surface are very difficult to render using traditional surface-based rendering strategies. Typical examples of these structures are fire, fractals, smoke or fog [KMJ$^+$06]. In this chapter a brief introduction into volume rendering is given and the most common techniques are introduced.

3.1 Basic Concepts of Volume Rendering

Techniques for displaying volumetric objects can be classified as *indirect* or *direct* volume rendering techniques, each of which has their own limitations and benefits regarding for example runtime performance or image quality.

3.1.1 Direct and Indirect Volume Rendering

Indirect Volume Rendering

The basic principle of indirect volume rendering is to segment and extract certain areas of interest from the volume and transform them into polygonal models. One example of an algorithm for this purpose is the *marching cubes* algorithm, which approximates a polygonal model from a voxel-based dataset [LC87]. This approximation of a surface can be rendered utilizing a standard rendering pipeline including all available illumination techniques. Feature extraction effectively means that only a subset of a dataset is taken into consideration. This results in a reduction of the data that needs to be processed during the rendering stage, which effectively increases rendering speed [Rit09]. The visualization of a volume by extracting and rendering isosurfaces may be a reasonable approach for particular data sets such as CT scans. The visualization of a human skull for example may give good results using a surface extraction. However for other types of data, e.g. fire simulation, this technique may be inappropriate, as an explicit surface may not exist in the original data [Rit09]. Another drawback of surface extraction is the fact that an intermediate preprocessing step has to be performed, which increases the overall complexity of the system. Moreover it is only possible to display the extracted surface; an additional display of surrounding information is not possible. For example it is not possible to perform a combined rendering including both iso-surface and transparent volume.

Direct Volume Rendering

The second category is *Direct Volume Rendering* (DVR). Using DVR techniques it is not necessary to extract a polygonal iso surface from the dataset. Instead the scalar values of the voxels are mapped to certain rendering attributes such as color and opacity. Thus DVR covers all techniques for rendering volumetric objects without an intermediate conversion to surface geometry [Levoy 1988]. During rendering all voxels are shaded according to a *transfer function* that associates distinct intensity ranges to distinct rendering properties. Depending on the transfer function different parts can be displayed or completely hidden in the visualization. The design of a suitable transfer function can be very time-consuming but also requires a lot of a priori knowledge about the data sets. Therefore it is crucial to offer interactive tools, that let the user dynamically update the relevant parameters and see the classification results directly.

3.1.2 Volume Rendering Integral

All volume rendering techniques make use of an *optical model* that describes how light interacts with certain media it travels through [KMJ+06]. The energy of light is described by its *radiance* I.

$$I = \frac{dQ}{dA_{\perp} d\Omega dt}. \tag{3.1}$$

Radiance is defined as radiative energy Q per projected unit area $dA_{\perp}$, per solid angle Ω and per unit of time t. While traveling through a medium, radiance gets altered due to the light's interactions with the medium. The three key interactions that affect the radiance of light are *emission*, *absorption* and *scattering* (EAS). The radiative energy of a light ray that interacts with a certain medium is affected by EAS. Optical models are used because the complete mathematical evaluation of this physical model of light transport is highly compute intensive and therefore not suitable for the domain of computer graphics. The most commonly used model in the context of volume rendering is the *emission-absorption model*. It describes the particle of a medium as light emitting and absorbing, while scattering effects are not taken into consideration. For a single ray, this model can be written in a differential form as,

$$\frac{dI(s))}{ds} = q(s) - k(s)I(s), \tag{3.2}$$

where emission is given by $q(s)$ and absorption by $-k(s)I(s)$. The parameter s defines the position along the ray. This equation is also known as the *volume rendering equation* for the emission absorption model [KMJ+06]. One possible solution for this equation is based on an integration along the direction of light flow from a given starting point $s = s_0$ to a given end point $s = D$. This integral solution is called the *volume rendering integral* $I(D)$.

$$I(D) = I_0 e^{-\int_{s_0}^{D} k(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} k(t)dt} ds. \tag{3.3}$$

The main purpose of DVR is the evaluation of this integral which computes the color of light that passes through a volumetric medium. A complete derivation of this integral is beyond the scope of this thesis. A comprehensive discussion can be found in [And95] and [MM04].

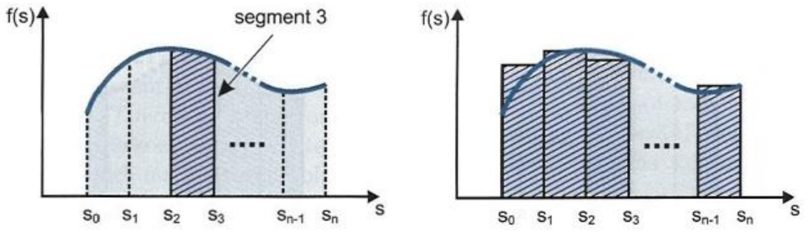

f(s)
segment 3
s
s_0 s_1 s_2 s_3 s_{n-1} s_n
f(s)
s
s_0 s_1 s_2 s_3 s_{n-1} s_n

Figure 3.1: Partitioning of an integration domain into separate intervals (left). Approximation of an integral by a Riemann sum (right) [KMJ+06].

Numerical Approximation

Generally it is not possible to evaluate the volume rendering integral analytically. Therefore the main goal of volume rendering is to provide numerical approximations that represent the exact solution as accurately as possible [KMJ+06]. One common technique is to split the integration domain of the volume rendering integral into n separate intervals. Each of these intervals is then approximated and the final result is determined by accumulating the intermediate interval results (Figure 3.1). The resulting approximation is known as *Riemann sum*. The volume rendering integral can be rewritten in a discrete form as,

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, c_0 = I(s_0). \tag{3.4}$$

This integral is the basis for all DVR techniques. It represents a good approximation to equation 3.3 and can therefore be used to generate two-dimensional images from three-dimensional data sets.

3.1.3 Data Representation

There are two different definitions of the term voxel. The one that is used as part of this thesis assumes that voxels are points in 3D space containing a coordinate and a corresponding scalar value. The scalar value of points that are located in between certain voxels is determined by trilinear interpolation. The second definition describes a voxel as a cubic region within 3D space. The scalar value of the voxel covers the whole interior of this cell. Following the first definition a voxel can be understood as a grid point within a uniform grid structure. The scalar value of each of these grid points can be distinguished by accessing the grid element at the respective coordinate of the voxel.

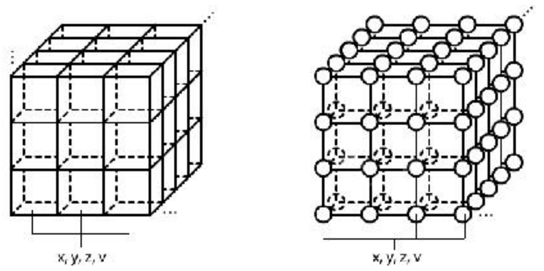


Figure 3.2: Two different types of voxel representation.

Therefore a single voxel can be described as a tuple *(x, y, z, v)*. The first three elements encode the spatial coordinate of the voxel. The corresponding scalar value is contained in the fourth element *v*. How this value has to be interpreted depends on the type of the dataset. Its intended meaning can be for example tissue density, temperature or seismic amplitudes. In the above example voxels serve as grid points of a *uniform* grid, where the distance between adjacent voxels is always the same. Conversely there are also non uniform-grids.

3.2 Volume Rendering Pipeline

3.2.1 Interpolation

During rendering an application will very rarely access the volume data at even grid points. Most of the time samples are taken which are located in between the voxels and the respective sample value has to be interpolated. During data acquisition a continuous signal is sampled at regular intervals and stored as a discrete voxel grid. Therefore a volume dataset represents the original signal in a discretized form. When sampling the data set at a certain position the original signal should be reconstructed as accurately as possible. Samples taken from in between voxels are interpolated and this operation serves as *reconstruction* of the original signal. The quality of interpolation is of great importance for the resulting image. There are many interpolation techniques, such as linear interpolations or cubic B-spline interpolation, that are relevant for volume rendering. Because it is used as part of this thesis, trilinear interpolation will be explained with the following example. The value of p in Figure 3.3 can be determined by a combination of multiple linear and bilinear

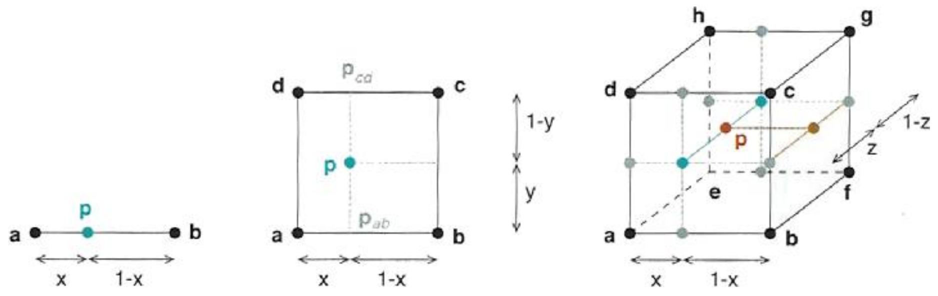


Figure 3.3: Illustration of linear (left), bilinear (middle) and trilinear interpolation (right) [KMJ+06].

interpolations. The value of p between point a and point b is calculated as,

$$f(\mathbf{p}) = (1-x)f(\mathbf{a}) + xf(\mathbf{b}). \tag{3.5}$$

A bilinear interpolation is calculated by two successive linear interpolations

$$f(\mathbf{p}) = (1-y)f(\mathbf{p}_{ab}) + yf(\mathbf{p}_{cd}), \tag{3.6}$$

where $f(\mathbf{p}_{ab})$ and $f(\mathbf{p}_{cd})$ are the results of two linear interpolations along the x direction,

$$f(\mathbf{p}_{ab}) = (1-x)f(\mathbf{a}) + xf(\mathbf{b}), \tag{3.7}$$

$$f(\mathbf{p}_{cd}) = (1-x)f(\mathbf{d}) + xf(\mathbf{c}). \tag{3.8}$$

In the same fashion trilinear interpolation can be calculated as,

$$f(\mathbf{p}) = (1-z)f(\mathbf{p}_{abcd}) + zf(\mathbf{p}_{efgh}). \tag{3.9}$$

A more extensive overview of interpolation and reconstruction techniques can be found in [KMJ+06].

3.2.2 Classification

A single voxel within a regular volume represents only intensity values but contains no information on the appearance of a certain element, such as color or transparency. This is because during data acquisition usually only the signal intensity is stored without further information on the appearance of a voxel. Therefore the optical material properties are not defined and must be adjusted manually. This process of mapping intensity values to visual rendering parameters such as opacity and color is known as *voxel classification*. Structures inside a dataset can be highlighted or completely hidden. For example voxels with a high intensity may belong to bones and can be rendered in white, whereas softer tissue such as fat or skin can be made transparent as it has a much lower intensity. In this way certain areas of interest can be selected based on the voxel intensity.

Transfer Functions

Such a mapping of intensity values to transparency is an example of a *one-dimensional classification* which is based only on scalar values since only intensity ranges are taken into consideration. The mapping that describes how certain intensity values are rendered is known as *transfer function*. These are generally implemented using some form of look-up table that contains optical attributes for all available intensity values. A basic opacity mapping can be encoded in a simple one-dimensional texture, however transfer functions can be two- or theoretically n-dimensional.

Even though one-dimensional transfer functions are the most widely used [CK09], they do have certain severe limitations. To overcome this, additional dimensions can be integrated into the transfer function domain leading to more flexibility during classification. The occlusion spectrum, which is a cross plot between signal intensity and respective occlusion, serves as a two-dimensional transfer function. In addition to the intensity, the occlusion values serve as a second classification dimension. By selecting appropriate regions inside the spectrum, structures can be separated - an operation that might not be possible using simple one-dimensional transfer functions.

3.2.3 Shading

Shading of a voxel is performed after it has been assigned color and opacity values. The final color of a voxel however is further dependent on parameters such as view direction, position and color of available light sources. There are certain models that describe how these parameters influence the appearance of a certain point. These models are known as *local illumination* models since they describe illumination of single surface points without taking global effects such as its ambient surroundings into consideration. Local illumination models calculate the color for a particular surface based on the normal vector that describes its spatial orientation. However in order to adapt local illumination techniques to volumetric datasets another metric is needed as they do not contain normals for its data elements.

Gradient Estimation

A good approximation for surface normals can be achieved by using the gradient of volume dataset which is defined as the vector that is perpendicular to the *isosurface* through that point. There are different approaches for calculating the gradient, including preprocessing and runtime techniques. Precomputed gradients generally result in high memory consumption as the gradients must be calculated and stored per voxel. An additional volume is uploaded to the GPU and during rendering gradients are sampled from this separate texture. Since GPU memory is mostly a scarce resource, this may be an unacceptable limitation. Another approach is to generate gradients during runtime. This so-called *on-the-fly* gradient estimation can be implemented using one of many available gradient operators.

The most common operator is the *central differences* operator that determines the difference between adjacent voxels based on,

$$x_d = \frac{1}{2}(\frac{f(x+h,y,z) - f(x,y,z))))}{h} + \frac{f(x,y,z) - f(x-h,y,z))))}{h}) \tag{3.10}$$

$$y_d = \frac{1}{2}(\frac{f(x,y+h,z) - f(x,y,z))))}{h} + \frac{f(x,y,z) - f(x,y-h,z))))}{h}) \tag{3.11}$$

$$z_d = \frac{1}{2}(\frac{f(x,y,z+h) - f(x,y,z))))}{h} + \frac{f(x,y,z) - f(x,y,z-h))))}{h}) \tag{3.12}$$

with h as the distance between two voxels and x, y, z as the coordinate of the central voxel [KMJ$^+$06].

The central differences operator approximates the directional derivatives in real time by taking six additional samples from neighboring voxels. The secant from the previous to the next voxel is then used as approximation for the gradient (Figure 3.4).

One important difference between precomputed and runtime methods is the way gradients are saved and sampled. Precomputed gradients are stored at regular grid coordinates and are interpolated trilinearly during runtime whereas runtime gradients are calculated on a per pixel basis inside the fragment shader. Since modern hardware has reached a performance level that is sufficient for runtime evaluation of gradients, this approach is generally

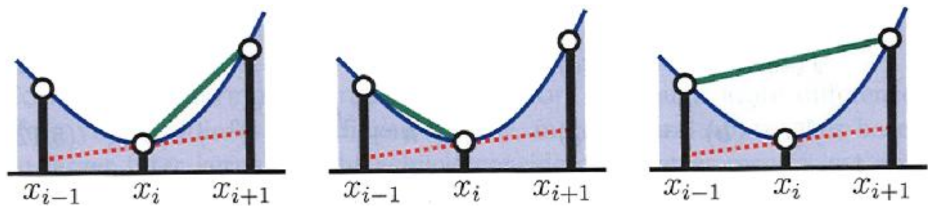


Figure 3.4: Finite differences for gradient estimation. Forward differences (left), backward differences (middle) and central differences (right) [KMJ+06]. The slope of the tangent (dotted red) is substituted by the slope of the secant (green).

preferred nowadays as it also results in higher image quality [KMJ+06].

Gradient-based shading is used best for volume datasets that have distinct and clear boundaries between material layers. Based on the gradient all common local illumination techniques can be applied. However, gradient-based techniques are very sensitive to high-frequency noise. In the context of seismic volume rendering this is problematic because acquisition is based on seismic waves and the datasets are generally of noisy nature with no distinct material boundaries [PBVG10].

3.2.4 Compositing

The basis for calculating the discretized volume rendering integral is the *compositing* scheme, which describes how samples that are taken along a ray are blended together to build the final pixel color. Although there are several different schemes for blending samples together, they all can be classified as either *front-to-back* or *back-to-front* traversal schemes depending on the direction of the viewing rays. The most common technique are front-to-back blending methods. Rays are cast from the viewing point into the volume and samples are taken at discrete positions along the ray. The blending formula for front-to-back traversal order is given by,

$$C_{dst} = C_{dst} + (1 - \alpha_{dst}) * C_{src}, \tag{3.13}$$

and,

$$\alpha_{dst} = \alpha_{dst} + (1 - \alpha_{dst}) * \alpha_{src}. \tag{3.14}$$

If rays are started at the back of the volume and are directed towards the camera then the blending formula is,

$$C_{dst} = (1 - \alpha_{src}) * C_{dst} + C_{src}. \tag{3.15}$$

Both approaches come with several benefits and drawbacks, so the traversal order should be selected depending on the application. One big advantage of front-to-back traversal is, however, *early ray termination.* The sampling of rays that have accumulated to full opacity can be canceled, because all portions of a volume that are located behind will be completely occluded, because the pixel is already totally opaque.

In addition to these two compositing schemes that are based on direction there are other methods such as maximum intensity projection (MIP) or accumulation methods (first, last, average etc.). Further information regarding this can be found in [KMJ^{+}06] and [MKG99].

3.3 Techniques for Volume Rendering

Volume rendering techniques can be classified as either *image-order* or *object-order* techniques. In this section for each group one example will be introduced, namely *texture slicing* and *volume raycasting.*

3.3.1 Texture Slicing

The concept of texture slicing is a very natural way of displaying volumetric datasets. For example, a CT scanner takes multiple stacked images of a human body. In discrete intervals two-dimensional sections are scanned and the measured density values are saved as pixel elements. In this way the continuous object is discretized and stored as a stack of images. A reconstruction of the scanned object can be achieved by placing multiple quad primitives closely next to each other. This group of quads is also called *proxy geometry.* In a basic implementation the position of the quad vertices are used to calculate texture coordinates at which the volume is sampled from. During rasterization each quad is then textured with a section through the volume with respect to its spatial coordinates. There are different ways for positioning the proxy geometry. The most common are *object-aligned* and *view-aligned* (Figure 3.5).

The advantage of basic texture slicing is its simplicity regarding the implementation as well as its runtime performance. Limitations however exist regarding the image quality, which often contains rendering artifacts at the edges of slice polygons. If texture slicing is implemented using two-dimensional textures, then only bilinear interpolation is available. On modern hardware this can be overcome by directly using three-dimensional textures that offer trilinear interpolation for sampling points located in between grid points. In

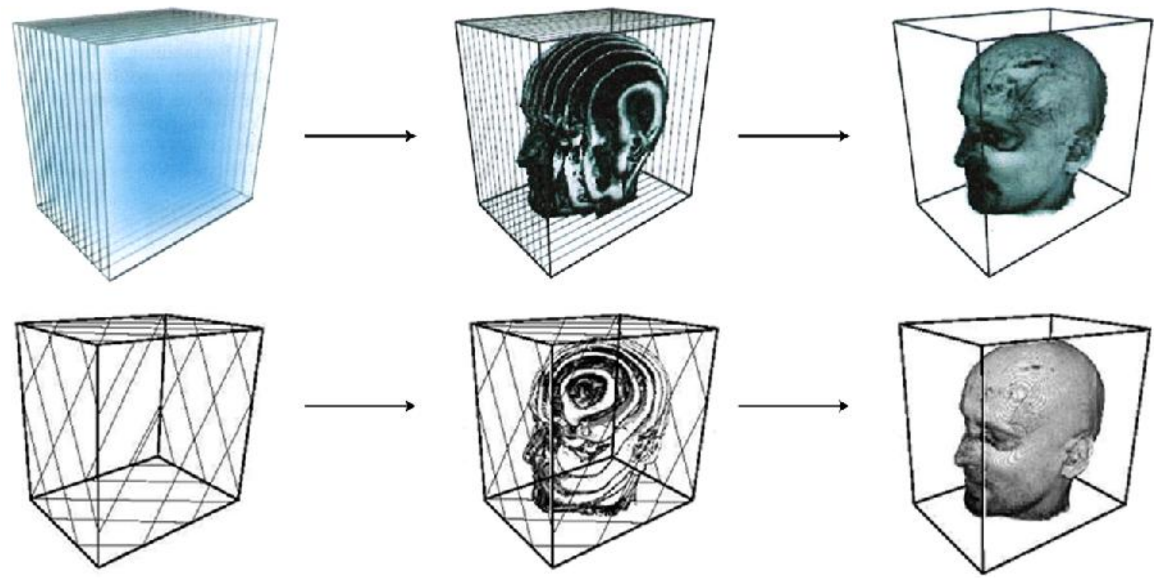


Figure 3.5: Multiple slices together form the final image. These can be either object-aligned (top row) or view-aligned (bottom row) [KMJ+06].

order to accurately display high frequencies in the data, proxy slices must be placed very close to each other, which results in a high amount of geometry that needs to be processed. Advanced rendering techniques such as adaptive sampling distances [MK07] can not be implemented easily into texture slicing algorithms.

3.3.2 Volume Raycasting

The basic idea of raycasting is to generate an imaginary ray for each pixel of the final image, which is illustrated in Figure 3.6. This ray is cast from the view point into the volume and is sampled at regular intervals. These samples are used to accumulate the color of the ray as it traverses through the volume. Color and transparency are applied at this point depending on the intensity of the voxel and the transfer function. Moreover, shading can be performed at this stage based on local illumination models and available light sources. Once the ray leaves the volume on the opposite side, the accumulated color is used as final pixel color. Raycasting is different to raytracing because the rays are not reflected at surface interfaces but instead are traversed through the whole range of the volume. Therefore the pixel color is determined as the sum of color and opacity values of all samples taken along that ray.

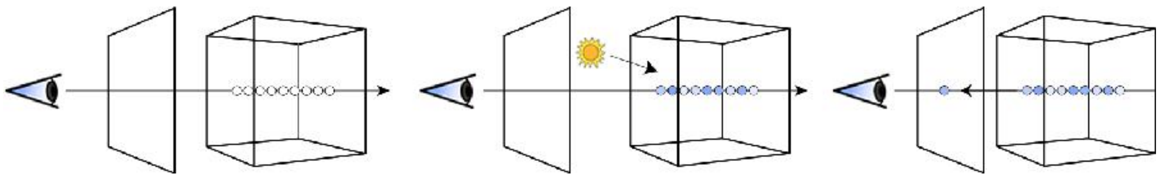


Figure 3.6: Steps performed during raycasting. Ray setup and ray traversal (left), sampling, classification, shading (middle) and compositing (right).

3.4 Seismic Volume Rendering

Volume rendering techniques are also used in the geoscientific domain to visualize data sets that are acquired through the use of seismic methods. In this context seismic amplitude values are mapped to rendering quantities. According to [PMM03] the main purpose of this approach is to get an overview of structural and stratigraphic features which are important indicators of the location of hydrocarbon traps. DVR is used to preview a seismic data set and identify areas of interest which are inspected in more detail in subsequent steps. Displaying a whole data set in one single image can be very helpful to gain an overview of a three-dimensional survey. Finer interpretation work is done afterwards based on potential hydrocarbon deposits that were identified using volume rendering.

In many visualization scenarios data sets can be rendered with some kind of default transfer functions for different types of data. These default functions serve as a good starting point for volume exploration and can be adjusted by the user during the rendering process. This process requires the knowledge of a priori models of the data. For example each CT scan of a human head will contain a certain amount of bones, skin and brain tissue. The distribution and the spatial relationships of these components to each other will vary with different CT scans, but all these structures are present. Unfortunately such a priori models do not exist for seismic data due to its strong variability [AMM03]. This means, that the distribution of structural and stratigraphic features differs with each data set.

Chapter 4

GPU Computing

Moore's law predicts that the number of transistors that can be placed on a single integrated circuit will approximately double every two years [Gor65]. However, recently this trend has experienced a slowdown because ever-increasing numbers of transistors and clock cycles per circuit cause a significant rise in temperature and energy consumption. As a consequence manufacturers of micro chips started to develop microprocessor architectures that contain multiple processing units to further boost the processing power of their chips. In this regard two main concepts are observable. The *multicore* concept follows the approach of using multiple processor cores, each fully implementing the x86 instruction set [DW10]. Multicore chips are designed to maximize the execution of sequential programs. In contrast to this, *manycore* chips are designed to maximize the execution of programs that are of a more parallel nature. Figure 4.1 illustrates the difference between both concepts.

The individual execution units of manycore systems are not multiple-instruction processors, but instead smaller single-instruction processors that are heavily multi-threaded and share their control and instruction cache with multiple other cores [DW10]. Examples of such parallel manycore processors are current Graphical Processing Units (GPUs) from Nvidia or AMD.

The Render Pipeline

GPUs were originally developed to accelerate the execution of graphics-related applications, which include a sequence of steps for rendering two-dimensional images from three-dimensional data. This sequence of steps is referred to as the *rendering-pipeline*, which is illustrated in a simplified version in Figure 4.2.

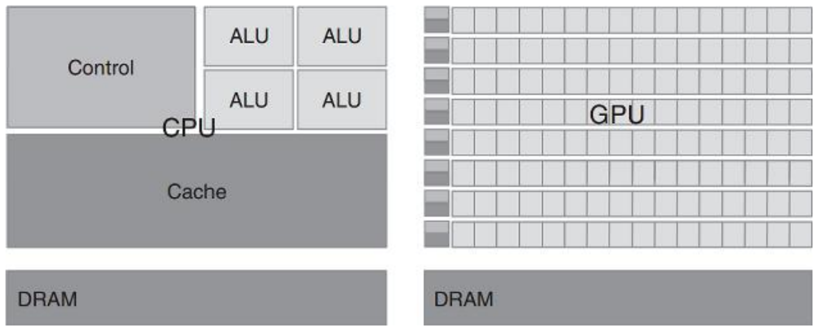


Figure 4.1: Difference between multicore (CPU) and manycore (GPU) [DW10].

The purpose of this pipeline is the processing of polygonal primitives into rasterized two-dimensional images that can be displayed on a computer screen. Application developers can manipulate the results of the different stages by writing small microprograms which are known as *shaders*.

As part of this thesis *vertex shaders* as well as *fragment shaders* are important. They can be used to control the effects of the vertex processing stage and the fragment processing stage respectively. The vertex processor performs per-vertex operations, which generally include operations such as linear transformations of the vertex position and/or its normal vector. Furthermore the input vertices are transformed from their local model coordinate system into the world space, then the camera space and finally into the screen space. Transformed vertices are subsequently grouped into geometric primitives in the *primitive assembly* stage [FK03].

After these per-vertex operations have been performed, the assembled primitives are propagated to the fragment processor. Each primitive is *rasterized* into a set of *fragments* where every fragment relates to a single pixel of the final pixel image. The generated fragments can be further processed by fragment programs that apply per-pixel operations, such as texturing or per-pixel illumination.

The results of the fragment processing stage are then blended into the frame buffer for compositing. In this last stage it is decided, if a single fragment will be visible on the screen or must be discarded due to occlusion [KMJ+06]. For the sake of completeness it must be noted that the pipeline of modern GPUs offers even more programmability in the form of *geometry shaders*, *tessellation control shaders* and *tessellation evaluation shaders*. However, since these are not relevant for the present thesis, the reader is referred to [Wol11] for more detailed information in this regard.

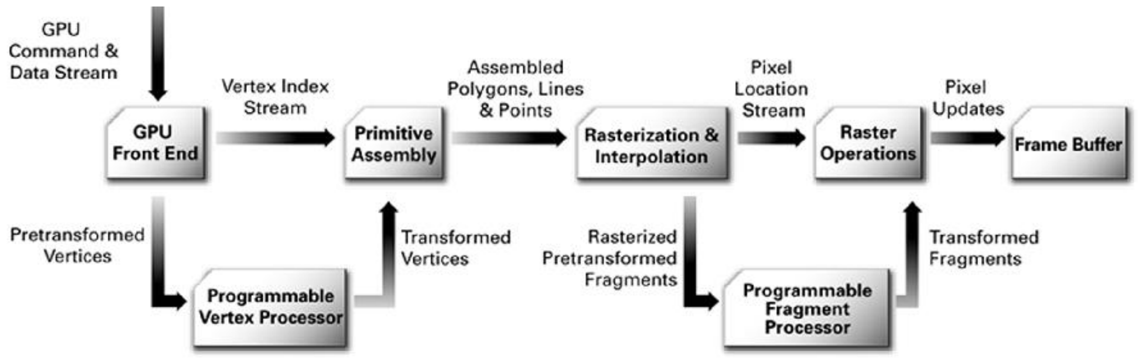

GPU Command & Data Stream
GPU Front End
Vertex Index Stream
Primitive Assembly
Assembled Polygons, Lines & Points
Rasterization & Interpolation
Pixel Location Stream
Raster Operations
Pixel Updates
Frame Buffer
Pretransformed Vertices
Programmable Vertex Processor
Transformed Vertices
Rasterized Pretransformed Fragments
Programmable Fragment Processor
Transformed Fragments

Figure 4.2: The Programmable Graphics Pipeline (Modified from [FK03]).

4.1 GPGPU

GPU computing or GPGPU (General Purpose Computation on Graphics Processing Units) covers a range of approaches to do general purpose computing on GPUs that can, but do not necessarily have to, be graphics-related. The basic principle is to combine multicore (CPU) and manycore (GPU) units to build a *heterogeneous* co-processing computing model [Gho12]. In this setup sequential parts of a program are executed on the CPU whereas portions of the application that can be parallelized are executed on the respective parallel processor. Therefore the benefits of heterogeneous systems lie in the combination of the strengths of both multicore and manycore technologies. The purpose of GPU computing is not to replace CPU computing. GPUs were originally designed for accelerating graphic operations. Using GPUs for other more general purpose applications, was difficult and usually forced the developer to detour via an existing graphics API, such as OpenGL or Direct3D. As a consequence, application logic had to be transformed into a format that could be executed in a graphics context, even though it was not graphics-related at all – an approach that was error-prone and that complicated application development [HN07].

4.2 Compute Unified Device Architecture (CUDA)

Nvidia's CUDA architecture was introduced in 2005 and aims at making GPU computing more accessible to developers. It is a programming model for modern manycore architectures. In order to access the computing power of modern GPUs developers had to transform their application logic into a format that could be processed within the graphics pipeline.

This means that parts of an application that were compute-intensive were processed in vertex and fragment shaders. However, shader syntax is limited and does not offer full access to all available hardware features. Even though CUDA applications are executed on the same hardware as graphic applications, more hardware features are accessible through the CUDA API. For example in contrast to fragment shader programs, a CUDA kernel can read and write to arbitrary positions in the global memory of the GPU [MRH10]. Although there are alternatives to CUDA such as DirectCompute or the vendor neutral OpenCL, at the time of writing, Nvidia's CUDA seemed to be the most mature and stable of these systems and was therefore chosen as GPU API for this thesis.

CUDA Syntax

Generally the syntax for writing CUDA applications is standard ANSI C with some extended keywords. These keywords are mainly needed for the compiler to distinguish whether given functions are supposed to be compiled for either host or device (Section 4.2.2) or what part of the device memory should be used (Section 4.3). A function that is declared with the keyword *global* will be executed as a kernel function on the device side but can be called from the host side. In contrast, the *device* keyword is used for functions that are only visible on the device side. This means that a function can only be called from within a kernel function, but not from the host process.

4.2.1 Fermi Hardware Overview

As stated above, multi- and manycore technologies are combined into a heterogeneous computing model that utilizes the strengths of both approaches. In this thesis the current *Fermi* generation is taken as an example of hardware architecture of modern GPUs. A Fermi GPU consists of multiple streaming processors which are supported by a second-level cache, the host interface and multiple DRAM interfaces (Figure 4.3). The *GigaThread* scheduler is responsible for distributing the available threads among the SMs as well as for redistributing the execution order of threads so that they get packed into the Fermi pipeline more efficiently [NVI09]. As can be seen in Figure 4.3, a Fermi GPU contains six 64-bit memory controllers that together result in a total memory interface of 348bit. The Fermi architecture can perform 16 double precision floating point operations per SM per cycle. Given, for example, a number of 16 SMs in the GTX 590, this results in 256 double precision operations per cycle. This is possible because Fermi GPUs work internally with

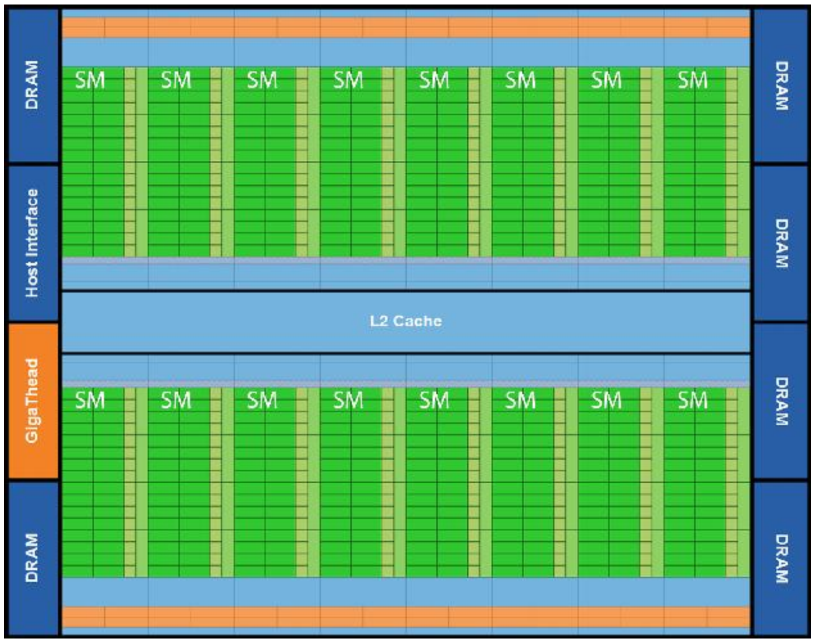


Figure 4.3: NVIDIA's Fermi GPU architecture consists of multiple streaming multiprocessors indicated in green [NVI09].

fused multiply add (FMAD) operations that can combine additions and multiplications into one single statement.

Fermi Streaming Multiprocessor

The main building block of Fermi GPUs is the *Streaming Multiprocessor* (SM), which consists of 32 CUDA cores each of which with a total number of 1024 registers. A single core has a dedicated integer ALU as well as an FPU that is capable of single and double precision floating point operations. The cores inside the SMs are SIMT (Single Instruction Multiple Threads) cores, which means they all execute the same instructions but on different parts of the input data. Furthermore, a single SM has 64KB available for shared memory and the L1 cache. How this amount of memory is split between shared memory and L1 cache can be selected by the developer (e.g. 48KB/16KB or 16KB/48KB). This feature leads to a higher degree of flexibility and gives the developer the option to increase the throughput when a task needs more or less L1 cache space [NVI09]. Each SM has further an 8-KB cache for constants that are stored in the device memory. A single SM can process up to 48 threads. The actual number of SMs differs with each chip version. Another component of an SM

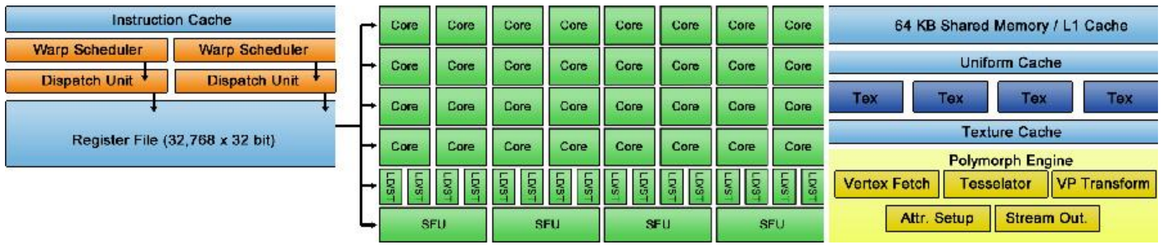


Figure 4.4: Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units (Modified from [NVI09]).

are 16 *LS* (load/store) units that manage access to the device memory. Besides that there are four *SF* (special function) units that are used for calculating special transcendental functions such as sine or cosine calculations.

Figure 4.4 shows an overview of one single streaming multiprocessor. Besides the actual execution cores, several control structures can be seen. At the top there is the *warp scheduler* and the *dispatch unit*. Together these two components are responsible for distributing *warps*, which are groups of 32 threads each among the 16 cores within the SM. For example, during program execution it can occur that certain threads have to wait for the result of a high latency operation such as a complex calculation or readings from device memory. The number of clock cycles it takes until a thread is ready for performing the next instruction is called *thread latency*. Other threads however might be ready for execution. The warp scheduler will always try to select warps that are ready for execution and can therefore try to hide thread execution latency. By always selecting warps that are immediately ready for execution, the hardware can be fully utilized, which increases the overall performance of a CUDA application.

4.2.2 CUDA Program Structure

A CUDA application consists of two main components. First there is the *host*, which represents a standard CPU-based system and processes the sequential part of an application, and secondly there is one or more *devices* that represent CUDA-enabled graphics hardware. During program execution the host calls so-called *kernel* functions that are executed on the device side. During the invocation of one of these kernel functions a certain number of *threads* is launched. These threads are organized in *blocks*; blocks in turn are organized in *grids*. This organization leads to a two-level hierarchy as shown in Figure 4.5. The size of a grid must be adjusted so that enough threads are generated to completely process the input data. The granularity of threads in a block as well as the granularity of blocks in a grid are parameters that can be adjusted manually, depending on the application and the data that is to be processed. For example, to define a kernel function that inverts the pixel values of a simple quadratic gray scale image with dimensions of 512x512 and a block size of 32x32 threads, the grid size can be determined as,

$$grid_{width} = \frac{input_{width}}{blockdim_{width}} + 1, \tag{4.1}$$

and,

$$grid_{height} = \frac{input_{height}}{blockdim_{height}} + 1, \tag{4.2}$$

which in this case results in grid dimensions of 17x17, which creates a total of 295,936 threads. Even though the image can be covered by a 16x16 grid, the additional block is needed in case the block dimensions can not be converted to a multiple of the input dimensions. The additional (in this example superfluous) block ensures that enough threads are always generated.

After a kernel function has been launched, the CUDA runtime system constructs the respective grid with the desired number of blocks and threads. Each thread block is distributed to and executed by one of the available streaming multiprocessors. The order of execution relative to other thread blocks can not be specified. This enables the hardware to schedule block execution in any order, in parallel or in series [Pet09]. This also allows the hardware to distribute thread blocks among the available execution resources. CUDA applications can therefore scale with increasing number of streaming multiprocessors. This ability to execute the same program code on hardware with different execution resources is known as *transparent scalability* [DW10].

Warps

Blocks that have been assigned to an SM are further subdivided into *warps* before the actual execution. Warps are the units of thread scheduling in SMs. A single warp consists of 32 threads and each thread block therefore has $N/32$ warps. In the Fermi architecture two warps from different thread blocks can execute concurrently [Pet09].

Each SM consists of a number of actual streaming processors. Thread blocks are further divided into warps because the hardware needs a certain thread granularity in order to

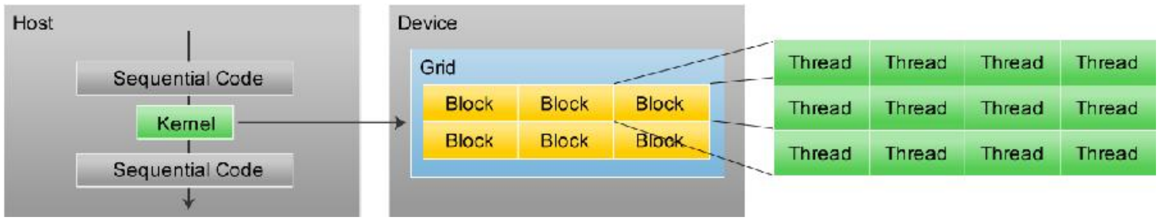


Figure 4.5: System structure of a CUDA application. Host side calls kernel functions. Kernel functions are executed on the device. Launched threads are organized in blocks. Blocks together form a grid (Modified from [DW10]).

efficiently execute long-latency operations such as accesses to global memory. If threads in a warp need to wait for the result of a long latency operation such as a complex calculation or access to global memory then the warp is not chosen for execution. Instead the SP processes another warp that is ready for execution. Executing other warps during latency of expensive operations is known as *latency hiding* and increases overall performance, because latency gaps are filled with other ready-to-process warps.

Kernel Functions

A CUDA grid consists of many small separate threads. During execution each of these threads processes instructions which are defined in a *kernel function*. Each thread in a grid executes the exact same kernel function. Therefore CUDA is based on the single-program, multiple-data (SPMD) parallel programming style [DW10].

In order to distinguish threads from each other the CUDA API provides certain predefined runtime arguments that can be used to identify threads but also to define which parts of the input data a thread has to process. A single thread in a grid is identified by two main values. The first value is the $threadIdx$, which defines the local position of a thread in a block. The second factor is the block index $blockIdx$ and the block dimensions $blockDim$, which locates the block inside the grid. These values are three-component struct variables that define the number of blocks in the grid, the number of threads in a block, the ID of the block within the grid and the ID of the thread within the block [NVI10a]. By combining these values, each thread can calculate its identifier as,

$$thread_x = blockIdx.x * blockDim.x + threadIdx.x \tag{4.3}$$

$$thread_y = blockIdx.y * blockDim.y + threadIdx.y \tag{4.4}$$

During execution of a grid, each thread needs to know which portions of the input data it is supposed to process. Since each thread has a unique ID, this portion can be calculated based on the thread identifier. Secondly, each thread needs to know where it should store its calculated result. In this case the thread identifier is also used to locate certain areas in GPU memory where threads can write their results to.

One thread can be seen as one iteration of a loop. For example, a function that inverts all elements within a gray-scale image would typically be implemented using two nested for loops. The single pixel coordinates are calculated sequentially. The same function can be rewritten as a CUDA kernel without any for loops simply by defining a grid that covers the whole input image and spawns one thread for each pixel. Within the kernel function, pixel coordinates can then be determined based on the location of the thread in the grid.

In the CUDA version, no for loops are needed anymore, as the function of the loop is now cast into the grid. To determine which pixel each thread has to process, the built-in variables *threadIdx*, *blockIdx* and *blockDim* are used. The final pixel coordinate can be calculated based on formula 4.3. While one single thread processes a single pixel, the kernel processes the whole image.

4.3 CUDA Memory Model

In a CUDA application, host and device have their own memory spaces. In this regard CUDA introduces its own memory model, which is illustrated in Figure 4.6. The host side of an application can only access the global and texture portion of the GPU memory. Therefore during program execution the host application must first allocate the needed amount of memory in device space. Subsequently the data needs to be transferred from host to device memory. Next the activation of one or more kernels takes place, which processes the input data according to kernel functions. Finally the data needs to be copied back from device to host memory space. To decide what parts of the memory model to utilize during program execution it is necessary to have an understanding of the benefits and drawbacks of the available memory types. Choosing the appropriate memory locations can have a great impact on the performance of an application as there are big differences in respect to access latency, the actual size of the address space as well as the scope of a memory location. Physically the memory types are either located in the device memory section or the cache memory section of the GPU. These are also sometimes referred to as

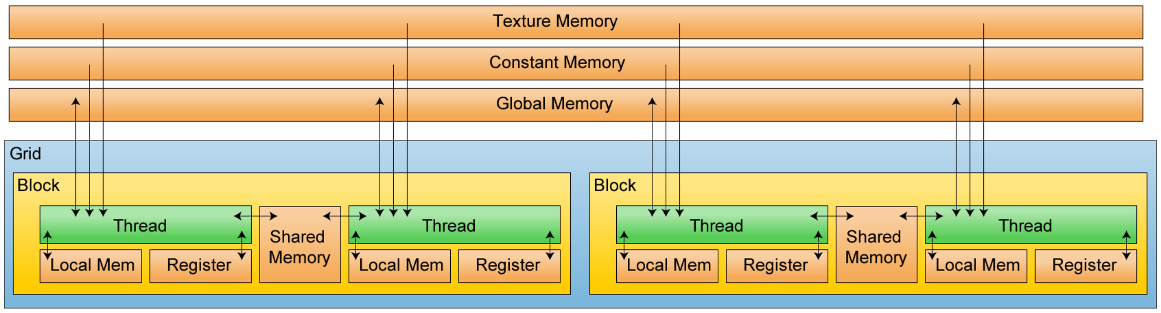


Figure 4.6: Overview of CUDA's memory model.

on-chip and *off-chip* memory sections respectively [DW10].

4.3.1 Off-Chip Memory

Global and Texture Memory

Global and texture memory can be accessed from the host side and represent the largest memory types. The actual amount of available global memory depends on the VRAM and is therefore different across graphic boards. However they are very slow in terms of access latencies. Texture memory is part of the global memory, but might be cached depending on the memory access pattern. It is read only for threads in a block, whereas global memory offers arbitrary read and write access. Global and texture memory are physically located in the device memory of the GPU and are therefore considered to be *off-chip* memories. Variables that are to be stored in either global or texture memory must be declared at global scope. That means the declaration takes place outside of the kernel functions at file scope and the lifetime of these variables is therefore the lifetime of the application. All threads that are spawned during program execution can access arbitrary global memory locations. The keyword for declaring a variable in global memory is *device*. It is the responsibility of the developer to manage the allocating and freeing of global memory resources.

Constant Memory

Constant memory is physically stored in off-chip device memory. However, accessing a variable in constant memory is considerably faster than global memory. This is due to the fact that the size of constant memory is limited to 64KB and it does not allow write access from within kernel functions. Therefore read operations to constant memory can be cached as it is guaranteed that the values will not be changed during the execution of kernel functions. All threads within a kernel will therefore see the exact same constant variables. Even though constant variables can be changed in between kernel executions this does not lead to an invalidation, because the variable will be "'locked"' as soon as the next kernel gets invoked. Constant variables must be declared at file scope using the *constant* keyword. Since constant memory is physically placed in device memory, it can be accessed directly by the host side. Just like global memory, constant memory has scope and lifetime of the application.

Local Memory

If a thread tries to use more registers than are available the surplus variables may automatically be relocated into the local memory that has the same high access latencies as the global memory. Developers therefore should try to avoid such a case by distributing register use more appropriately among the available resources. One big difference to other types of memory is that a placement of data into the local memory can not directly be influenced by using respective identifiers as is possible with other memory types. Variables will be stored automatically in local memory if a thread consumes too much register space or if array indices are not known at compile time. In these cases the compiler will choose to place an array into the local memory address space. Just like register variables, local memory variables have the scope and lifetime of the thread.

4.3.2 On-Chip Memory

Register

This is the default memory location where all variables are stored that are declared within the scope of a kernel function. These include scalar, automatic and also array variables as long as they are declared with constant indices so that the size of the array can be determined at compile time. Data stored in registers is private to a single thread and can not be accessed from an outer scope. Register variables are only alive as long as the thread is executing. Registers are the fastest of the available memory types regarding its access latencies. If a specific application demands storage of these variables in other memory locations this can be defined by one of the respective identifier keywords. Registers are part of the on-chip memory section and there is only a limited number of registers that can be used per block. This also means that registers can not be accessed directly from the host side.

Intra Block Shared Memory

Shared memory has a very limited address space but allows very low latency access (100 times faster than global memory) [NVI10b]. This on-chip memory can be shared between all threads inside one block and enables what is called *intra-block communication*. However, threads from different blocks can not communicate. Accesses to shared memory can be as fast as access to hardware registers and it is physically located in the cache memory. Shared memory variables must be explicitly defined by the developer by using the keyword *shared*. The actual amount of shared memory depends on how a developer has distributed the available 64 KB among the shared memory and the L1 cache. Even though shared memory variables are declared at the same scope as register and local variables, they have a scope and lifetime of the block. Therefore contents of shared memory can still be accessed after individual threads have finished execution. The effective use of shared memory can drastically increase application performance, as it offers much faster access times. Since blocks generally only need to perform executions on a subset of the input data, a very common technique is to load relevant subsets into the intra block shared memory. Calculations are then performed only by accessing the shared memory. Subsequently the results are copied back into the global memory. In this way global memory traffic can be decreased, which is beneficial for the overall performance of an application. This is also very important in the context of this thesis, since the algorithm for computing the occlusion makes heavy use of shared memory.

Chapter 5

Accelerated Occlusion Computation

In this chapter the new algorithm for computing occlusion is introduced. It was developed to overcome the major bottleneck of the original version, which was the generation of the occlusion. It was necessary to experiment quickly with different settings and parameters and then compare the results in a preview environment. The new algorithm has two major advantages. First it reduces the number of texture samples by building occlusion based on differences between adjacent voxels. Secondly it is implemented in a parallel fashion and executed on modern parallel processors such as GPUs.

5.1 Occlusion Computation from a Volumetric Dataset

5.1.1 Brute Force Sampling

The original pipeline used a brute force approach for the precomputation of the occlusion information. To process a given voxel all its neighbors that are located within the sampling area had to be accessed. Obviously this approach resulted in a huge number of texture samples, but also in a huge number of calculations. In addition to taking samples from the volume texture it is also necessary to sum up the neighboring intensities and multiply the result with a weighting factor. Figure 5.1 illustrates the sampling area for one single voxel with a sampling radius of 4. To calculate the occlusion for the green center voxel, all surrounding voxels have to be sampled. The single intensity values are added to the

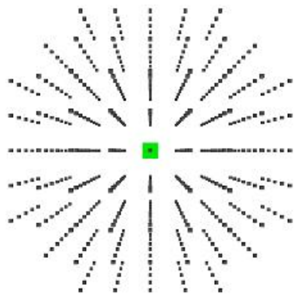


Figure 5.1: Sampling area for a single voxel with radius 4.

occlusion of the target voxel with the weighting of $1/N$ where N is the total number of samples taken. In this example, occlusion can be understood as a weighted local histogram for a given voxel with a single bin.

This sampling pattern has to be performed per voxel. For a relatively small cubic volume with dimensions of 512^3 and a sampling radius of 25 the total number of texture samples N needed for computing the occlusion can be calculated as,

$$N \approx a^3 \frac{4}{3} \pi r^3 \tag{5.1}$$

$$N \approx 512^3 \frac{4}{3} \pi 25^3 \tag{5.2}$$

$$N \approx 8.784.529.755.548 \tag{5.3}$$

where a is the volume extent in one dimension and r is the radius of the sampling area. Such a brute force approach is very slow especially with increasing sampling areas or volume dimensions since it has a complexity of $O(n^3)$.

5.1.2 Cached Occlusion Computation

The new algorithm is based on the observation that the sampling areas of two adjacent grid points overlap and share many common voxels. That means if the sum for a specific voxel is calculated, then a big part of the adjacent sum is already built. If this value can be cached and reused for adjacent voxels, then the new sum can be determined by building only the difference between two neighboring voxels. In this case only voxels need to be considered that are located on the borders of the sampling area. To calculate the occlusion for the

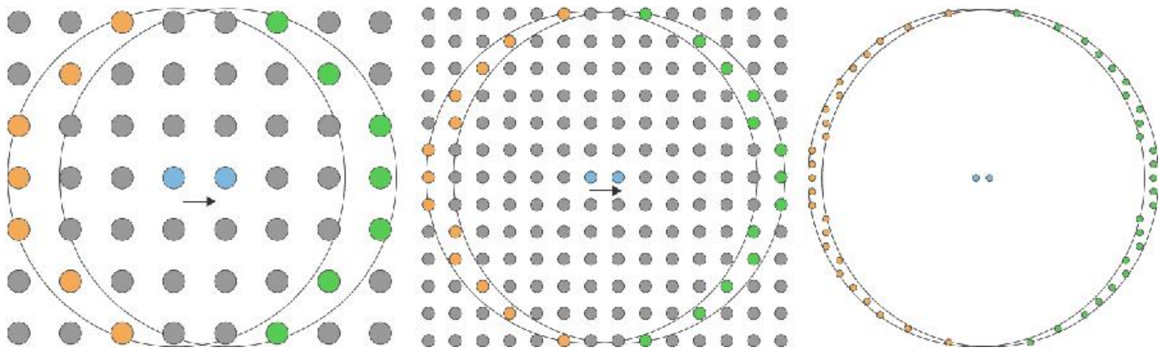


Figure 5.2: Adjacent occlusion can be calculated under consideration of border voxels. Subtraction voxels are indicated in orange, addition voxels are indicated in green.

blue voxel in Figure 5.2, texture samples are only necessary for border voxels. The new sum can be calculated by subtracting the orange and adding the green voxels. This figure also illustrates that this approach scales very nicely with increasing sampling areas (middle and left).

Sampling Mask Structure

For the sake of simplicity the current implementation does not use a circular/spherical sampling area, but a rectangular one instead. However, spherical sampling areas can still be simulated by applying a distance factor that reaches 0 once the distance between two grid points exceeds a certain radius. Therefore the current sampling area is defined by two border lines indicated in orange and green in Figure 5.3. By moving this sampling mask through a volume, the occlusion value $o_{(x,y,z)}$ for one single voxel can be calculated as,

$$o_{(x,y,z)} = o_{(x-1,y,z)} + \frac{(add - subt)}{N}, \tag{5.4}$$

where N is the number of voxels inside the sampling mask and *add* and *subt* are the summed intensities of the respective border lines. This example assumes that the sampling mask is moved along the x-axis, however this is not mandatory. Depending on the volume dimensions it may be beneficial to move the sampling mask along a different axis.

In order to further reduce the number of texture samples this algorithm is divided into four phases. Each phase differs in the way texture samples are taken and also in the way the resulting occlusion value is saved.

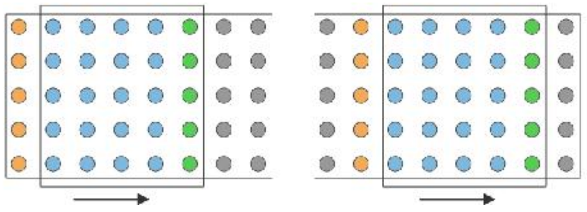


Figure 5.3: Rectangular sampling mask defined by border voxels is pushed through the volume.

Phase 1 – Build initial sum

In the beginning the sampling mask is positioned slightly outside the volume coordinates (Figure 5.4, left). From this point on the mask is pushed into the volume and as long as the position index of the sampling mask is outside of the volume ($idx < 0$) the resulting occlusion value is only written into a separate occlusion cache. This is necessary because once the sampling mask reaches the volume border ($idx = 0$) the occlusion sum for the very left group of voxels is correctly initialized. For comparison if a starting index is chosen that is located exactly on the border of the volume ($idx = 0$) it is necessary to explicitly sample the initial occlusion sum in order to build differences in subsequent iterations. This would be possible but would require a second sampling strategy that gets executed only once to build the initial sum. By positioning the sampling mask outside the volume there is no need to distinguish between different sampling patterns. It is only a matter of checking the index to decide if the result is only to be cached or additionally saved into the results volume.

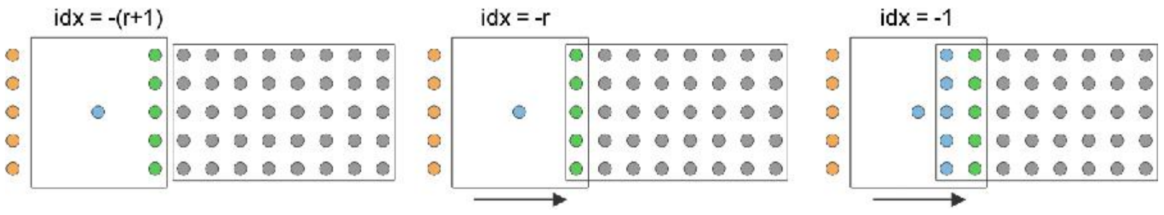


Figure 5.4: Phase 1 of sampling algorithm.

Phase 2 – Addition only

This phase starts as soon as the sampling index has reached the border of the volume (Figure 5.5, left). From this point on the calculated occlusion value is not only written into the cache for reuse in the next iteration, but also saved inside the final occlusion texture.

It would have been possible to omit the cache and read the previous occlusion value from the result volume, but this would mean one additional access to texture memory. As long as the subtraction border is still outside the volume the occlusion value can be determined by only sampling the addition border.

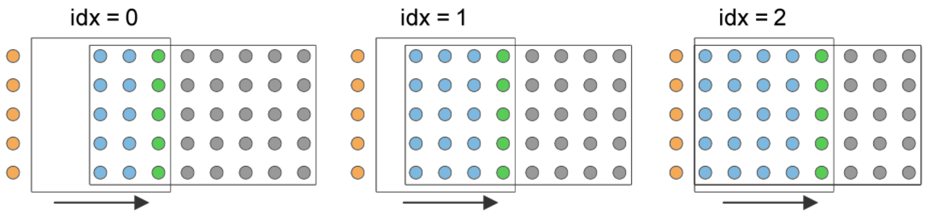


Figure 5.5: Phase 2 of sampling algorithm.

Phase 3 – Addition and Subtraction

This is the phase that gets executed the most during occlusion computation. Both sampling lines are inside the volume border. Thus texture samples need to be performed for addition but also for subtraction voxel (Figure 5.6).

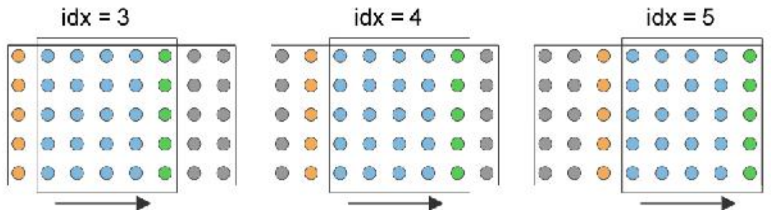


Figure 5.6: Phase 3 of sampling algorithm.

Phase 4 – Subtraction only

In the last phase (Figure 5.7) the addition border has left the volume and the remaining occlusion values can be calculated by subtracting the orange voxels from the cached occlusion value. Texture samples are only necessary for the subtraction border.

5.2 Parallel Implementation

The above technique was implemented in a parallel fashion using the CUDA programming environment, which is a parallel computing architecture developed by NVIDIA. The relevant

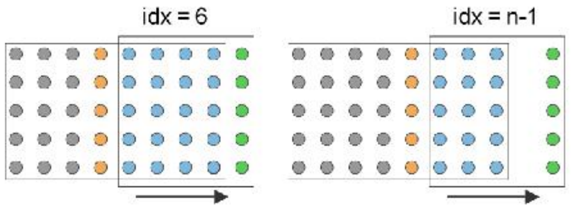


Figure 5.7: Phase 4 of sampling algorithm.

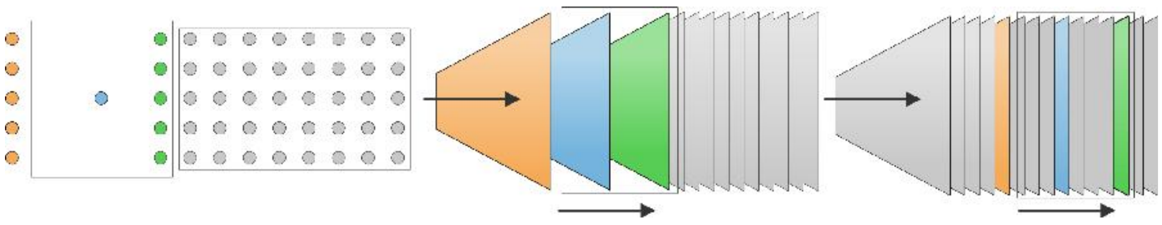


Figure 5.8: Extended sampling mask is defined by two border planes.

basics regarding CUDA programming were covered in chapter 4.

5.2.1 Application to Volumetric Data

As stated above, the sampling area is defined by two border lines; one samples the new voxels that get added and the second samples those voxels that get subtracted from the cached occlusion value. In order to use this concept for calculating occlusion for a three-dimensional volume, it is necessary to extend the sampling mask as shown in Figure 5.8. Instead of two border lines, the new extended sampling mask is defined by two border planes. Both of these planes have the same dimensions as one slice through the volume.

In order to correctly calculate the initial occlusion that is needed for the start slice, the sampling mask is positioned outside of the volume. Sampling is now performed by moving the sampling mask through the volume and caching / saving the results according to the above-described phases. The cache in this case is a two-dimensional array and can be thought of as located *in between* the active planes. Each array element represents one voxel.

Another observation that can be made by looking at Figure 5.8 is that during each iteration there are two active slices. In the current parallel implementation each of these slices is further subdivided into blocks. Each block consists of a two-dimensional group containing a number of 32^2 threads. According to CUDA's thread organization hierarchy

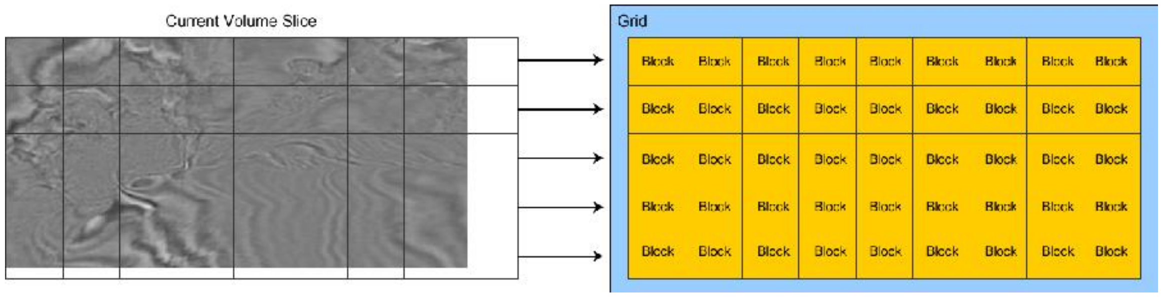


Figure 5.9: One slice of the volume is further subdivided into blocks. Depending on the slice dimensions blocks may partially be located outside the slice.

(Figure 4.5), this means that we create one *grid* per border *plane*.

For example, consider a slice with a size of 1000x500 voxels and a block size of 32x32 threads. Then the total number of *blocks* that is required in order to process all elements of one slice can be written as,

$$N_{blocks} = (\frac{vw}{bw} + 1) * (\frac{vh}{bh} + 1) \tag{5.5}$$

$$N_{blocks} = (\frac{1000}{32} + 1) * (\frac{500}{32} + 1) = 512 \tag{5.6}$$

The total number of *threads* generated for one slice is thus,

$$N_{threads} = N_{blocks} * bw * bh \tag{5.7}$$

$$N_{threads} = 512 * 32 * 32 = 524.288 \tag{5.8}$$

In this example one thread per voxel is spawned. However it is worth noting that the total number of threads may differ from the number of voxels of one slice. This is because in all cases where the volume dimensions is not a multiple of the block dimension, there will be an overlap at the borders to ensure that the whole slice area is covered by threads (Figure 5.9, left). 524,288 threads per slice might seem very high at first sight, but CUDA threads are very lightweight compared to CPU threads and require only a fraction of cycles to setup and manage. Furthermore it is recommended to use a certain number of threads in order to fully utilize the processing capabilities of the hardware (see chapter 4).

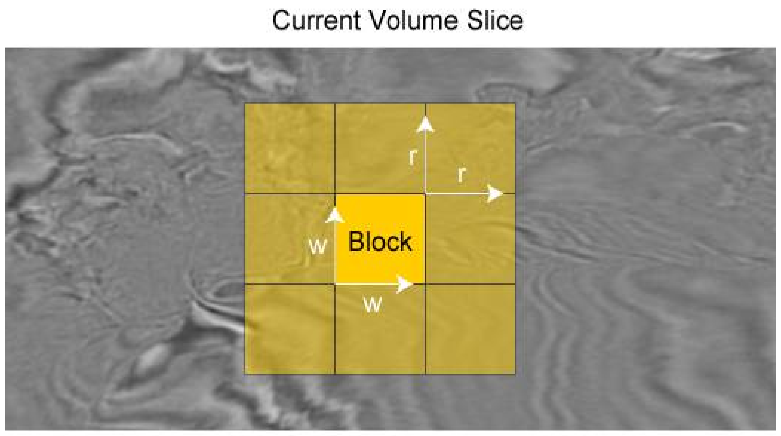


Figure 5.10: Blocks only require a subset of the current slice.

5.2.2 Shared Memory Loading

It can be seen in (Figure 5.10) that each block only requires a certain part of the current slice. All voxels that are not located within the yellow area are not relevant for the occlusion computation of the yellow block. The area that contains the required voxel for one block can be calculated as,

$$A = (2 * r + w)^2, \tag{5.9}$$

where r is the radius of the sampling area and w is the width of the current block. It can be seen that a lot of voxels from area A will be used multiple times during occlusion computation for this block. The texture access patterns for neighboring voxels will be almost identical except for the outermost samples. The total number of samples that are needed for processing one block are,

$$N = A * sv, \tag{5.10}$$

where nv is the number of voxels inside the current block and sv is the number of samples per voxel defined as,

$$sv = (2 * r + 1)^2. \tag{5.11}$$

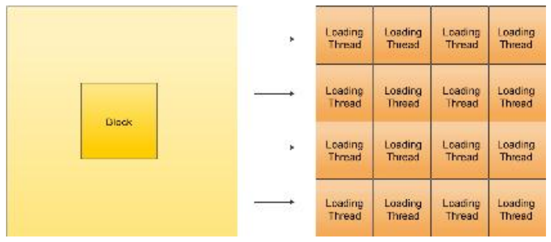


Figure 5.11: Shared memory loading for one block. The sub-slice is divided into 16 parts that are loaded by separate threads.

In order to reduce the number of accesses to texture memory, each block loads the relevant sub-slice into its shared on-chip memory. Since the Fermi architecture contains 16 load store units per streaming multiprocessor (see chapter 4), each sub-slice is further partitioned into 16 equally sized blocks that are loaded by separate threads. This means that before occlusion computation is performed, 16 threads of a block transfer the voxels from area A into the shared memory and apply the visibility function. All other threads in the block are blocked until the transfer is completed. In this way all available load store units are utilized, which is sometimes referred to as *memory-level parallelism* [BCK11].

At this point the visibility mapping can be applied directly as it will be the same for all threads inside the current block. To sum up this means that each thread calculates the occlusion for one voxel. This is done by reading the occlusion from the previous voxel from the occlusion cache and applying the difference between the addition and subtraction border. During the occlusion computation itself, there are no more samples taken from the global texture memory, because all the relevant preclassified voxels are held within the low latency on-chip memory. On completion each thread writes its result into the cache and also the final occlusion volume texture.

5.2.3 Algorithm and Data Flow Overview

An overview of the flow of data during occlusion computation for a volumetric dataset is given in Figure 5.12. In the beginning a transfer of the original intensity data from host to device memory takes place. All subsequent steps are then executed in device space.

After the initial transfer the actual occlusion computation starts. As stated earlier one grid is created per border plane. There are two borders during each iteration. The part of the program that gets executed by each grid is the blue marked area in the above figure.

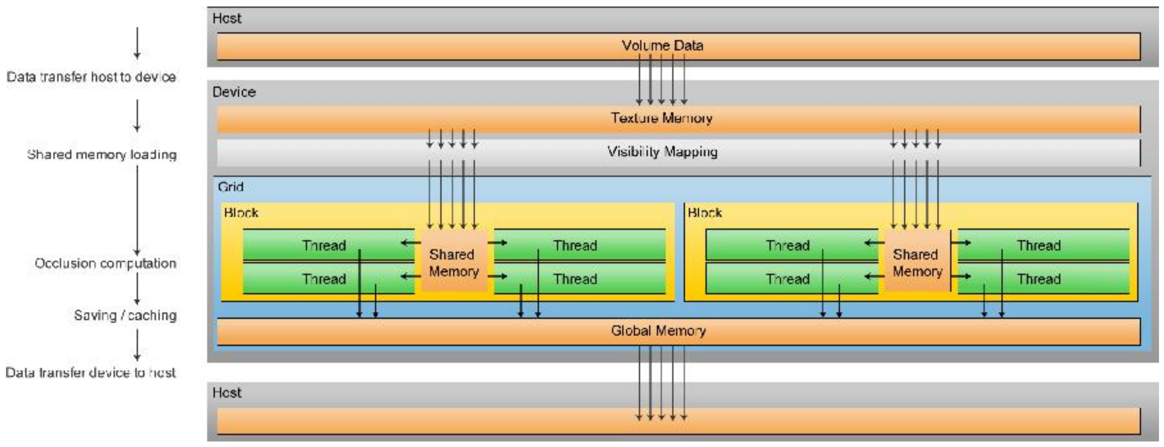


Figure 5.12: Flow of data during occlusion computation.

At the beginning of grid execution all blocks use 16 loading threads to load the relevant sub slice into its shared memory and apply the visibility function. In the subsequent occlusion computation only the shared memory needs to be sampled, which is up to 100 times faster in terms of access latencies [NVI10b]. Also the visibility function has already been applied at this point, hence there are no redundant calculations. On completion the resulting occlusion volume is transferred back into host space for further processing. The major differences of this approach to the first version are the following. First and foremost a reduction in complexity is achieved by only taking texture samples for voxels that are located on the border of the sampling area, which leads to fewer texture samples and also to fewer computations. Furthermore the new algorithm is now executed in a parallel fashion utilizing very fast on-chip memory.

Chapter 6

Adaptive Mapping Selection

A Visibility mapping determines the amount of occlusion contribution for different intensity values. It can be used to emphasize or hide certain intensity ranges and has a great influence on the quality of the resulting histogram. Tests (see section 8.3) indicated that specific mapping functions result in good contrast for some regions while others might require different mappings to gain a similar contrast.

In order to select an optimal mapping automatically, an adaptive method was developed. The current approach is based on local histograms that are generated in a preprocessing step utilizing a slightly modified version of the existing parallel algorithm (see chapter 5). Local histograms are saved in a simple database. The amount of data that has to be stored separately is controlled by selecting appropriate bin sizes as well as limiting the process to certain intensity ranges.

A temporary occlusion spectrum is generated for different combinations of histogram and mapping functions. This is done in terms of a convolution between histogram and mapping function and can be performed very quickly. The quality of a mapping is then subsequently rated by analyzing the variance of means inside the spectrum. An overview of the algorithm is shown in Figure 6.1. The three distinct steps for determining an optimal mapping are the following:

1. Local histogram generation for a given intensity interval of interest
2. Combination of suitable mapping function with the local histograms
3. Evaluation of the quality of a mapping by analyzing the variance of means in the resulting spectrum

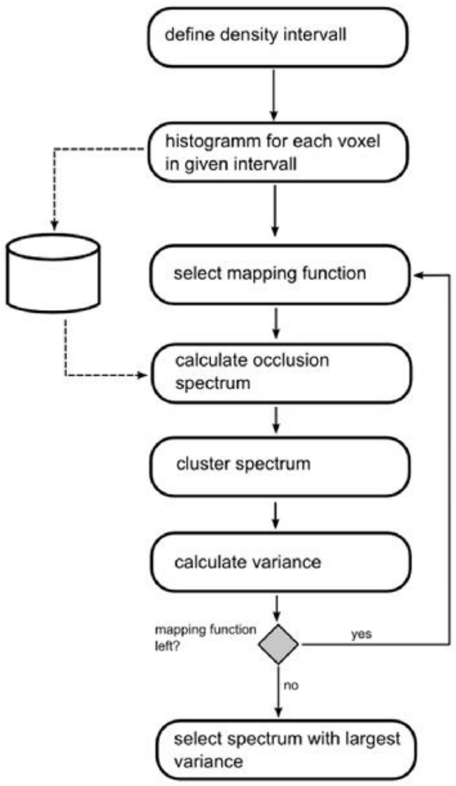

define density intervall
histogramm for each voxel in given intervall
select mapping function
calculate occlusion spectrum
cluster spectrum
calculate variance
mapping function left?
yes
no
select spectrum with largest variance

Figure 6.1: Steps performed for selecting an optimal mapping function for occlusion generation.

6.1 Local Histogram Generation

To calculate the occlusion value for a given voxel a certain neighborhood around its grid point must be taken into consideration. The occlusion is determined by building the weighted average of intensities of neighboring voxels. The weighted average is equivalent to the centroid of the local histogram for a given voxel [CK09]. This local histogram encodes the information of a local neighborhood of points in 3D space. In order to speed up the execution, local histograms are generated and saved separately once for the whole dataset. In subsequent steps the different visibility mappings are only applied to the histogram in terms of a convolution and not by sampling and weighting the original volume texture anymore. However, saving a histogram per voxel will require quite a lot of additional memory. Consider a histogram with 256 bins, each of which is encoded with one byte. In this case the additional amount of memory required is 256 bytes per voxel. Since volumetric datasets

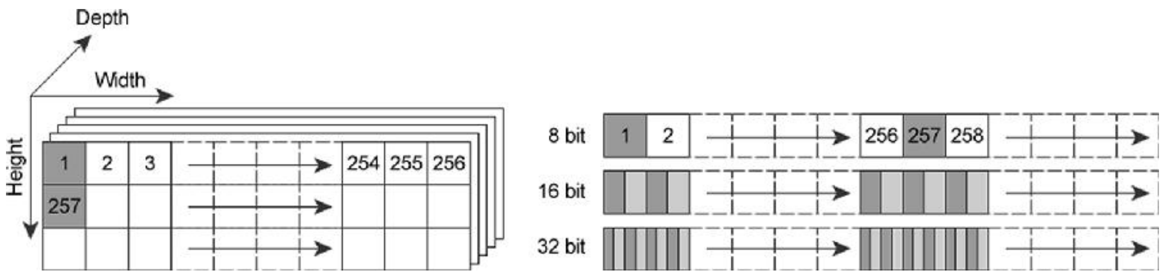


Figure 6.2: Volume data is stored in a sequential memory layout. Each image element is encoded with 1, 2 or 4 byte depending on the data format.

already require a substantial amount of memory a multiplication of the data by a factor of 256 clearly is not an option. Two factors can be adjusted to overcome this. By choosing a more appropriate *bin size*, the amount of additional memory can be reduced by a factor of $256/nbins$. Secondly, local histograms are only generated for certain *intensity intervals* of interest and not for the whole volume. Depending on the interval borders and the data distribution of the dataset this might further reduce the amount of memory.

6.1.1 Sequential Histogram Generation

Since local histograms are sampled from a cubic neighborhood for a given voxel with spatial coordinate x, y, z and sampling radius of r the total number of taken samples is,

$$N = (2 * r + 1)^3. \tag{6.1}$$

Volume data is stored in a linear memory layout as illustrated in Figure 6.2. The first byte location of the relevant neighborhood within the linear memory layout can be found by,

$$b = (h_{src} * w_{src}) * z_{min} + (w_{src} * y_{min}) + x_{min}) * num_{bytes} \tag{6.2}$$

where num_{bytes} is the number of bytes per component depending on the data format, h_{src} and w_{src} are the original volumes height and width. The minimum extent point of the sub volume can be determined by subtracting the vector (r, r, r) from the voxel's spatial coordinate. The maximum extent is found by adding (r, r, r) to its coordinate. The side length of the sub volume is defined as,

$$len_{xyz} = max_{xyz} - min_{xyz} \tag{6.3}$$

After reading the first len_x number of bytes starting from b, which in this case represents the first line of the sub volume, it is necessary to skip dx number of bytes. After reading all lines of the first sub-volume slice dy number of bytes need to be skipped. The next iteration starts over with the next sub-slice. The number of bytes that have to be skipped in each iteration are determined by,

$$dx = (w_{src} - len_x) - 1, \tag{6.4}$$

and,

$$dy = (h_{src} - len_y) - 1 \tag{6.5}$$

In this fashion all relevant voxels that are located inside the sampling range can be addressed directly inside the linear memory, which increases loading speed.

6.1.2 Parallel Histogram Generation

Even though calculating the local histograms is performed only once, it is still a major bottleneck in this algorithm. In its basic form, the algorithm for computing a histogram in three dimensions has a complexity of $O(n^3)$ since the number of samples taken is directly related to the extent of the sampling area in each of the three dimensions. To overcome this the generation of local histograms was integrated into the parallel occlusion algorithm that was introduced in chapter 5. The most important property of this algorithm was a reduction of complexity from $O(n^3)$ to $O(n^2)$. This was achieved by reusing results from one iteration and building the new occlusion by calculating the difference between adjacent voxels. Texture samples are only necessary for voxels that are located on the borders of the sampling area. This is possible because the neighborhood of two adjacent voxels will be mostly equal besides the opposite borders of the sampling domain. The same observation is also valid when it comes to the generation of local histograms, as the process of collecting the histogram information is very similar. Neighboring voxels need to be sampled and their value is then distributed among the bins for the local histogram. Local histograms between adjacent voxels are therefore also equal besides the values that are located on the borders of the sampling area. Instead of explicitly sampling the complete neighborhood for a given voxel, a local histogram can be constructed by the histogram of the previous voxel and a so-called *difference histogram*.

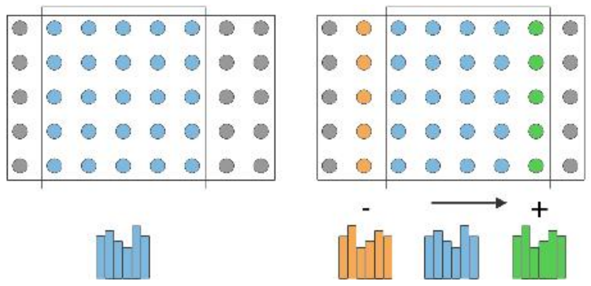


Figure 6.3: Local histogram can be generated as difference histogram.

To achieve a meaningful comparison the generation of local histograms has been implemented in three different ways. The first is a basic CPU version that builds the local histogram using the brute-force strategy. The same approach has also been implemented for execution on the GPU using CUDA. The third version is integrated into the new tiled parallel algorithm.

6.2 Mapping of Visibility Functions

The visibility function (or visibility mapping) determines the amount of occlusion contribution for different intensity values. They can be used to highlight or hide certain intensity ranges and thus have a great impact on the quality of the resulting occlusion crossplot.

In order to accelerate the proces of applying visibility mappings to a volume data set the process is split into multiple steps. After the local histograms have been generated, visibility mappings can be applied in terms of a convolution between histogram and mapping function. This means that the visibility function is not applied when building the neighborhood of a voxel, but instead at a later stage. This effectively increases execution times, because the sampling step with complexity $O(n^3)$ has to be performed only once. The complexity of performing the convolution between histogram and visibility mapping is much lower as it is only performed in a two-dimensional space.

6.2.1 Mapping Groups

Correa already suggests a couple of different mapping functions in his paper [CK09]. However in this context seismic data is being dealt with and therefore additional mapping groups have been included into the new algorithm. Since seismic data is generally normally

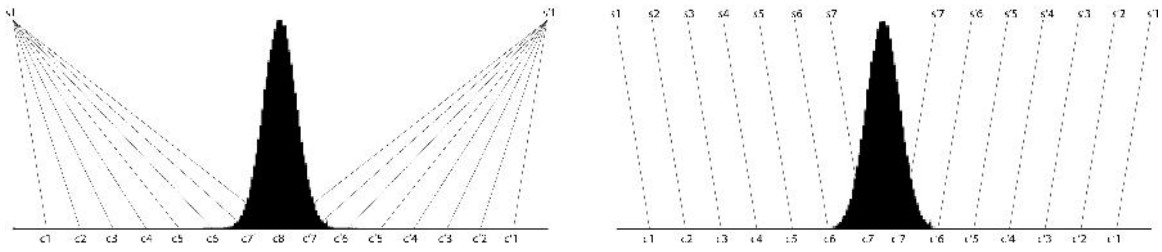


Figure 6.4: Two examples of mapping groups. On the left side the intersection points c1 – c8 are diversified, the upper intersection points s1 is constant. On the right in addition to the intersection points c1 – c7 also the upper intersection points s1 – s7 are diversified, leading to much steeper lines around the middle peak.

distributed, it is necessary to apply a mapping that accounts for this symmetrical data distribution. In addition to the linear and truncated ramp functions from the original paper we integrated *seismic ramps*. Seismic ramps are symmetrical visibility functions. For each group of seismic ramps a set of variants is defined as can be seen in Figure 6.4.

6.3 Determination of Optimal Mapping

Within the next step, a temporary occlusion spectrum is generated for each of the predefined mapping functions using the pre-calculated local histograms. This is done in terms of a convolution between the histogram and the mapping function and can be performed very quickly. A clustering of the resulting occlusion spectrum is performed afterwards. The quality of a mapping is then subsequently rated by analyzing the variance of means of the specific cluster centers.

6.3.1 k-Means Clustering

Clustering is done based on an implementation of the *k-means* algorithm, which easily finds the cluster center of a given set of similar points. It works by defining an initial number k of cluster centers which are distributed randomly among the set of input points. Subsequently each point is assigned to the cluster whose center is nearest to the point. Generally this can be done using any kind of distance function but in this case a simple geometric distance is used. After all points have been assigned the center points of the clusters are updated. Afterwards all points are again assigned to the clusters but this time based on the updated cluster centers. This process of point assignment and center update is repeated until a maximum number of iterations has been reached or the cluster center do not change anymore.

6.3.2 Variance Analysis

The quality of a visibility mapping is rated based on the variance of the centers after the clustering has been performed. As was described in section 1.2 a good occlusion spectrum is vertically expanded and therefore contains a high dynamic range of occlusion values. In contrast, a very flat occlusion spectrum does not offer a reasonable range of occlusion values and approaches a one-dimensional classification space. Therefore a good occlusion spectrum consists of multiple clusters that are spread all over the available range because this maximizes the likelihood of structure separation [CK09]. Thus a mapping that maximizes the variance of the cluster centers can be rated as good, whereas mappings that result in less variance can be considered as bad. For each available mapping function a temporary occlusion spectrum is generated. After the spectrum has been clustered, the variance of the centers is calculated. The mapping that results in the overall maximum variance of means is then choosen.

Chapter 7

Occlusion Editor Application

Evaluating a classification technique for volume datasets requires interactive tools, that let the user experiment with all different parameters. Besides the tools that are custom made for the occlusion it is crucial to offer basic tools in order to enable a direct comparison. For control purposes it is essential to offer well-known features such as opacity mapping or volume cutting planes. Cutting planes are especially important as they are used to represent survey lines, which are an integral tool and their importance in the context of seismic volume exploration can not be overemphasized [KMJ$^+$06]. Furthermore, image statistics must also be provided in the form of amplitude histograms. These histograms serve as visual reference when applying transparency to distinct intensity ranges or when defining a visibility function for occlusion generation. This chapter introduces the editor application that has been developed to support the evaluation process of occlusion-based classification of seismic datasets.

7.1 System Overview

The editor application is written in standard ANSI C++. The preview rendering is based on the *OpenSceneGraph* API. It is used for transferring data to GPU memory and for establishing the required OpenGL context. The GUI portion of it is based on the *wxWidgets* Toolkit. The current version is implemented for Windows only, but there are no serious reasons why the application could not be ported to other platforms. However this includes only operating systems. Because the parallel algorithm is implemented in CUDA, the application requires Nvidia hardware, because CUDA is not vendor-independent. To allow for the editor to be run on non-Nvidia hardware, the project is split into two libraries. The editor itself is independent of the CUDA requirements and can thus be used on other hardware and operating systems. The CUDA implementation that is maintained as a separate project however must be executed on Nvidia devices. In such a scenario a connection could be established via a webservice for example.

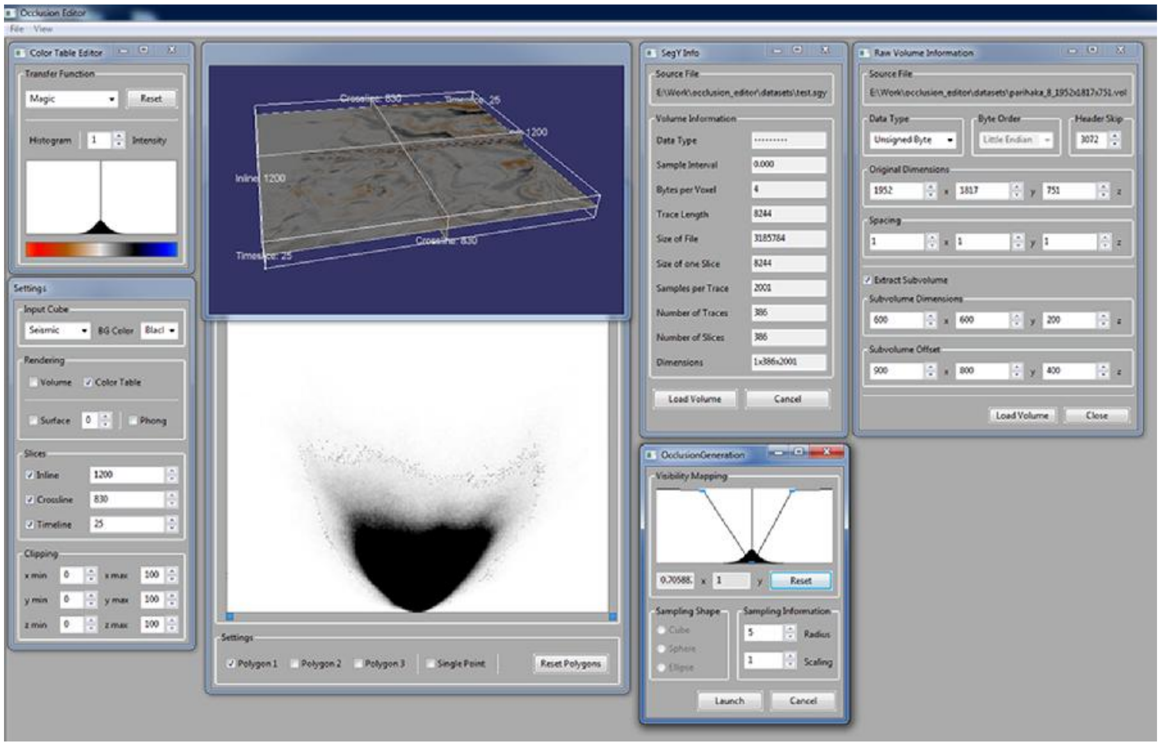
User Interface

Within the editor all relevant functions are integrated into one interface. This includes all data handling functions such as volume import and export, interactive definition of mapping functions, occlusion generation based on the parallel CUDA implementation as well as preview rendering. The renderer features a volume raycaster that is implemented in GLSL and contains local illumination based on a central differences scheme and opacity mapping. In order to allow for a better evaluation of the results, the renderer also offers interactive features such as cutting planes, volumetric clipping as well as a transfer function editor.

7.2 Volume Processing

Data Formats for Volumetric Seismic Data

Seismic data can be stored as a regular binary volumetric dataset. Therefore it was crucial to support the importing of binary RAW streams. Besides the binary raw data another very common exchange format for seismic data is SEG-Y [MWN01]. Work flows are very important when it comes to seismic interpretation and thus support for SEG-Y was integrated since most of the big seismic interpretation software packages offer import and export of this format. However only import of seismic data is not enough. Additionally the application must support the export of volumetric datasets. Export functionality is essential to evaluate the results in more sophisticated seismic interpretation packages. This opens up new possibilities for further analyzing seismic occlusion by comparing the results with other seismic attributes (see section 2.3.3). As a consequence the application that was developed as part of this thesis supports importing and exporting of either RAW or SEG-Y formatted data.



Transfer Function: Magic
Histogram: 1 Intensity

Input Cube: Seismic BG Color: Blacl

Rendering: Volume [✓ Color Table]; Surface 0; Phong

Slices:
[✓ Inline] 1200
[✓ Crossline] 830
[✓ Timeline] 25

Clipping:
x min 0 x max 100
y min 0 y max 100
z min 0 z max 100

Settings: [✓ Polygon 1] Polygon 2 Polygon 3 Single Point

Source File: E:\Work\occlusion_editor\datasets\test.sgy

Volume Information:

Data Type	**********
Sample Interval	0.000
Bytes per Voxel	4
Trace Length	8244
Size of File	3185784
Size of one Slice	8244
Samples per Trace	2001
Number of Traces	386
Number of Slices	386
Dimensions	1x386x2001

Source File: E:\Work\occlusion_editor\datasets\parihaka_8_1952x1817x751.vol

Data Type: Unsigned Byte
Byte Order: Little Endian
Header Skip: 3072

Original Dimensions: 1952 x 1817 y 751 z

Spacing: 1 x 1 y 1 z

[✓ Extract Subvolume]

Subvolume Dimensions: 600 x 600 y 200 z

Subvolume Offset: 900 x 800 y 400 z

Visibility Mapping: 0.70588 x 1 y

Sampling Shape: Cube Sphere Ellipse

Sampling Information: 5 Radius; 1 Scaling

Figure 7.1: Interface of the occlusion editor application.

RAW Datasets

Seismic data that is formatted as RAW is stored in a linear address system on disc, which is illustrated in Figure 6.2. One common problem in the field of seismic volume rendering is the data size, which very often exceeds the size of available video memory. Of course this can be overcome by implementing an out-of-core data management, but this was not the focus of this work. Therefore another approach has been implemented which is referred to as *subvolume extraction*. If the original dataset is too large to fit into video memory, the user can define arbitrary subvolumes during the loading stage. This is currently only supported for RAW formatted data, however there exists a workaround that also allows subvolume extraction for SEG-Y files. RAW formatted volumes are expected to be stored in a sequential memory layout. This is required to efficiently calculate the byte positions of the subvolume while loading. Byte positions of subvolumes are calculated in a similar fashion as the local histograms during the adaptive mapping process (see section 6.1.1). This is because a local histogram can be considered as a subvolume from the whole dataset, which is spatially defined by the coordinate of the center voxel and the radius of the sampling area.

SEG-Y Datasets

The SEG-Y format is a standard binary format for storing seismic data. A typical SEG-Y file has the following structure. The first 3200 bytes are reserved for a human readable textual header. This header contains general information on the datasets, such as survey name, date of data acquisition and so on. After that there follows a 400 byte binary record. These two sections together form the main header section of the dataset. These header may be followed by optional textual file headers. The rest of the dataset contains the actual volume data. These are encoded as seismic traces (see chapter 2), each of which is preceded by a separate 240 binary header. An overview of a typical SEG-Y file is shown in Figure 7.2. The standard for SEG-Y allows the data to be formatted in a variety of ways. The demo application however supports only 16-bit SEG-Y data, which can be exported from any common seismic interpretation software.

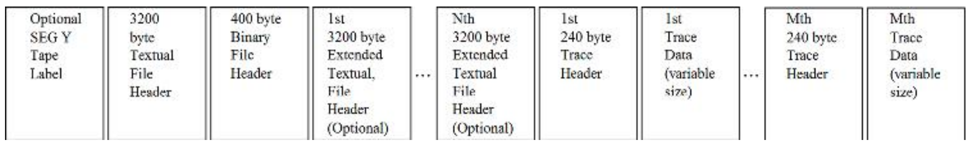

Optional SEG Y Tape Label
3200 byte Textual File Header
400 byte Binary File Header
1st 3200 byte Extended Textual, File Header (Optional)
...
Nth 3200 byte Extended Textual File Header (Optional)
1st 240 byte Trace Header
1st Trace Data (variable size)
...
Mth 240 byte Trace Header
Mth Trace Data (variable size)

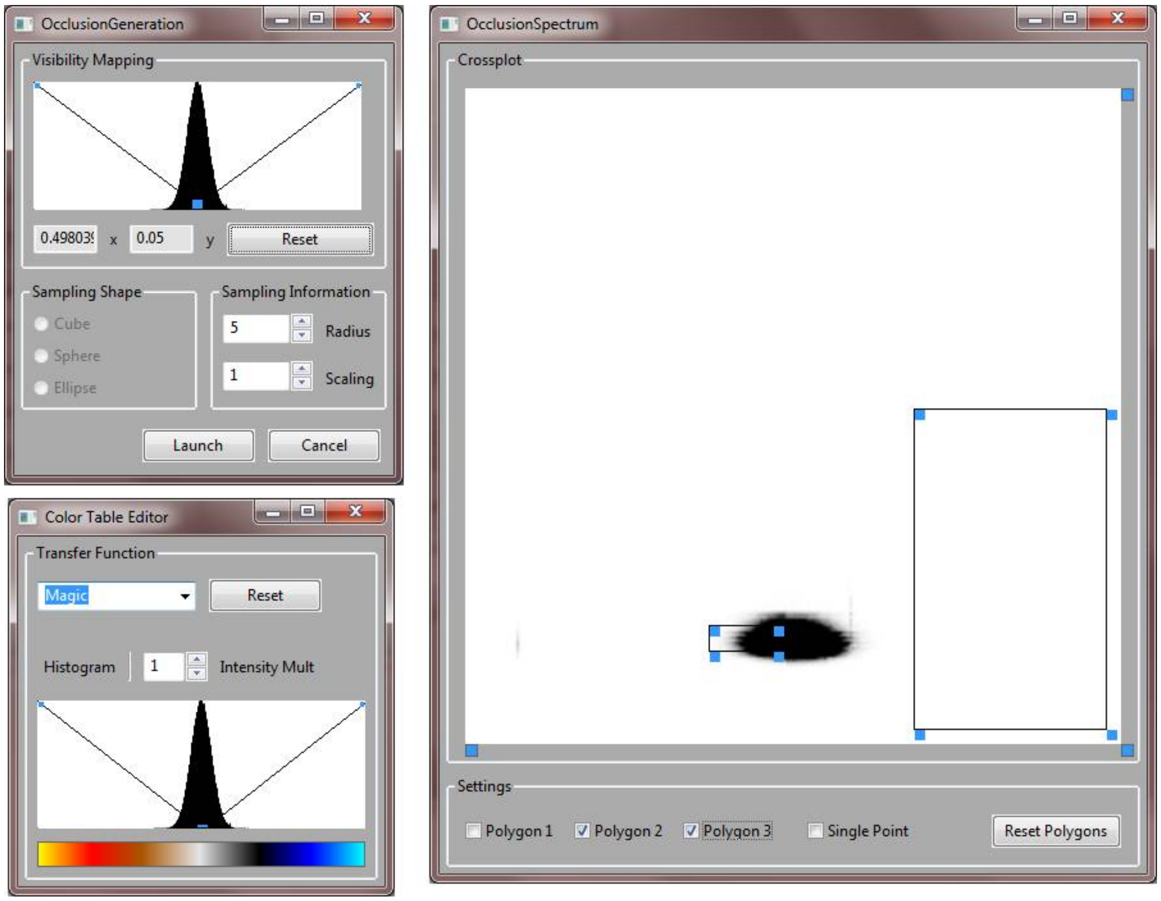
Figure 7.2: File layout of a SEG-Y file [MWN01].

7.3 Occlusion Generation

Occlusion is calculated in a parallel fashion using the algorithm that has been discussed in chapter 5. This algorithm is integrated into the editor application and occlusion processing can therefore be performed without switching the application context. This simplifies the work flow and is more user friendly than splitting occlusion generation and preview of the results into separate tools. Generating occlusion inside the application requires certain steps. First a dataset has to be loaded. Secondly the user needs to define a visibility mapping that serves as an additional transfer function to map intensity values to occlusion values. This step can be performed using the custom visibility function editor. To aid the process of defining a suitable visibility mapping it is crucial to directly see the relationship between mapping and the current dataset. Therefore a tool was developed that lets the user directly paint the visibility mapping over the amplitude histogram of a given dataset (Figure 7.3). After a dataset has been processed the results can be analyzed by directly rendering the occlusion volume. However, detailed information on the quality of the resulting spectrum can only be gained by looking at the crossplot between original signal amplitudes and the respective occlusion. This two-dimensional histogram is the actual occlusion spectrum. It is displayed in a separate dialog window that enables the user to interactively show or hide regions from the crossplot (Figure 7.3).

7.4 Preview Rendering

The preview section of the editor application is based on a volume ray casting with a front-to-back traversal order. It features local illumination with on-the-fly gradient estimation based on a central difference scheme (see section 3.2.3), iso-surface rendering and early ray termination. It has a custom transfer function editor for selecting different color tables as well as interactive editing of the opacity function (Figure 7.3). To concentrate the view to a specific part of the volume, binary clipping is integrated that completely removes



0.498039 x 0.05 y

○ Cube
○ Sphere
○ Ellipse

5 Radius
1 Scaling

Magic

Histogram 1 Intensity Mult

☐ Polygon 1 ✓ Polygon 2 ✓ Polygon 3 ☐ Single Point

Figure 7.3: Custom widgets. Visibility function editor (top left), color table editor (bottom left) and two-dimensional transfer function editor (right).

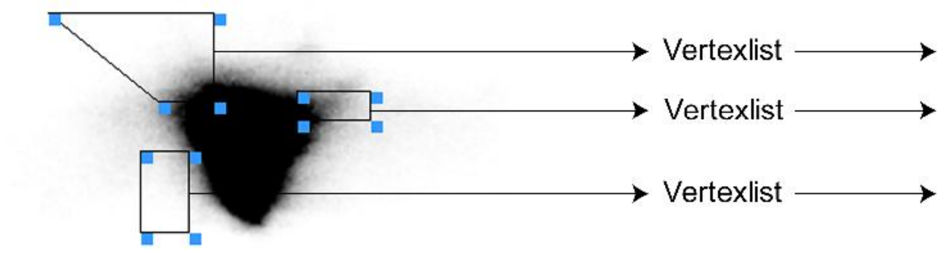


Figure 7.4: Selection polygons define visible regions of the spectrum.

certain parts of the volume from the rendering. In addition to the actual volume rendering, cutting planes are available that represent two-dimensional cross sections through a volume. The spatial orientation of these slices can be either as inline, crossline or timeline (axial, coronal, sagittal). Furthermore, arbitrary regions can be selected inside two-dimensional transfer functions by manually placing and controlling *selection polygons*.

Selection Polygons

In order to hide certain parts of the occlusion spectrum during rendering, it was necessary to develop tools which let the user interactively select regions of interest. The area that is visible in relation to the selection inside the crossplot is defined by selection polygons. To determine if a particular voxel is to be rendered a point in polygon intersection test needs to be performed, that is based on the following algorithm.

A point that is cast along a ray will possibly intersect a given polygon. Depending on the direction of the ray and the shape and position of the polygon there may be zero or multiple intersections between ray and polygon. Each intersection will either mean that the ray enters or exits the area of the polygon (Figure 7.5). An odd number of intersection means that the ray is inside the polygon and an even number means that it is outside of the polygon. This technique can be used to effectively decide if the user has clicked inside a selection polygon. This intersection test is actually used twice within the application. First it is used to determine whether the user has clicked and thus selected the inner area of a selection polygon. Secondly it is used inside the shader. Since the shader only knows the intensity values from either the original volume texture and the occlusion it must further determine if a given voxel is located within a selected area as defined by the selection polygons.

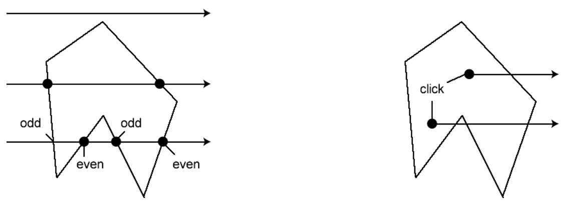


Figure 7.5: Intersection of rays with arbitrary polygons.

Volume Raycasting

The raycasting is implemented in GLSL. Most of the instructions are processed by the fragment shader. The vertex shader is only used for calculating the camera position in world space and propagating vertex positions to the fragment shader. The camera position can be determined simply by inverting the model view matrix and multiplying it with the vector $v(0,0,0,1)$. Based on the camera and vertex position, rays can be calculated for each single fragment. The ray direction is the normalized difference between vertex position and camera position. Rays are cast into the volume and sampled at regular intervals. For performance reasons the original seismic volume and the generated occlusion volume are packed into a single file. Seismic amplitudes are stored in the luminance channel and the occlusion is stored in the alpha channel. Therefore during ray traversal only one sample needs to be taken from texture memory which contains the intensity and occlusion in the first two channels. These values represent the two axes of the occlusion spectrum.

Visibility Test

In a next step it must be determined whether or not a given combination of intensity and occlusion is visible or invisible according to the user-defined selection polygons. Therefore the two sample values are queried against a list of polygons that make up the user-selected regions. This is the same algorithm that is also used to decide if a user has clicked inside a selection polygon (Figure 7.4). A given combination of intensity and occlusion is only visible if it is located inside one of the selection polygons.

If the current sample passes the visibility test it is further classified by querying the transfer function. If local illumination is activated, then the gradient for the current position is estimated by a central differences scheme and used as normal replacement in a standard Phong illumination model. The resulting color is then blended into the color buffer. Ray traversal continues with the next iteration until the color buffer has accumulated to full opacity or until the ray exits the volume.

Inverse Workflow

In order to better establish a relationship between occlusion and seismic regions inside the volume it was necessary to develop an inverse selection technique. The user can click a voxel inside the volume and see the relevant parts highlighted inside the spectrum. This enables a bidirectional relationship between the original volume dataset and the resulting occlusion. This tool is also processed at runtime and can thus be used interactively, which is crucial for evaluating the usefulness of this approach.

Empty Space Skipping

Depending on the transfer function and the selected region inside the occlusion spectrum the amount of data, that is visible of a volume, may be low. In this case the renderer has to perform a lot of work by only skipping voxels that are completely transparent. Furthermore all rays have to pass through the whole volume and will never accumulate to full opacity. Early ray termination has no effect in this situation. Empty space skipping (ESP) is an effective solution to this problem. It delivers information to the renderer that can be used to skip regions of the volume that do not contain meaningful data and thus accelerates the rendering. Studies have shown that typically only a fraction of the actual volume is visible [KMJ+06].

Datastructures for ESP

In order to perform ESP an additional data structure must be introduced, that subdivides the volume into discrete bricks and encodes the minimal and the maximal occurring amplitude within that block. Different data structures can be used for this purpose; one of them is an octree. In this very context such a data structure is called *min/max octree*, since each cell contains in addition to the spatial resolution and position also statistical information on the intensity values inside each cell.

The octree is generated recursively. The volume is split into eight equally sized child cells. Subsequently the minimal and maximal intensity values inside each cell are determined by sampling all voxels in the cells. If these values are beyond a certain threshold, the cell is again subdivided into eight blocks, as long as a maximal tree depth is reached, or if a given

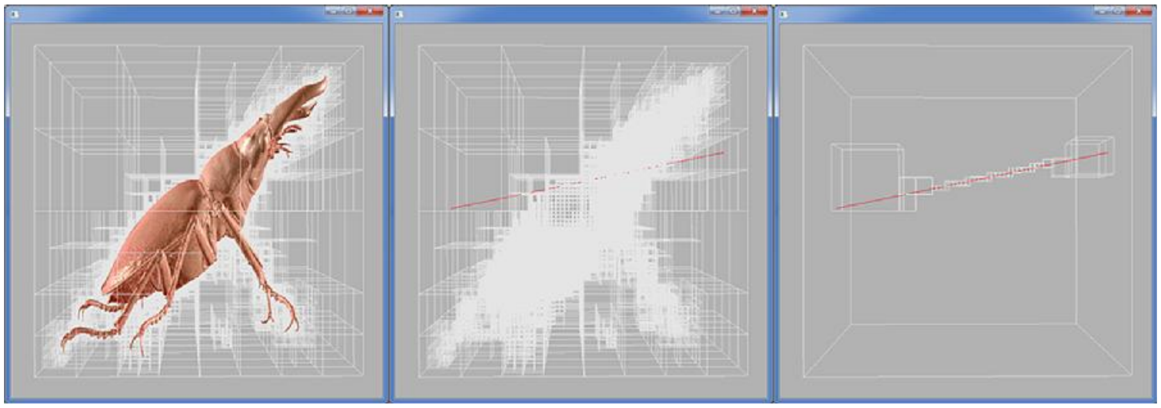


Figure 7.6: Empty space skipping in occlusion editor application. Sampling distance can be adjusted based on min/max octree to skip empty regions in the volume that do not contain any meaningful data.

cell does not pass the threshold test.

During rendering this data structure can be queried and the sampling distance inside the ray traversal loop can be adjusted adaptively. When a ray enters a cell, it can query the data structure for the min/max values of that cell. If these values are visible according to the transfer function, the sampling of the ray is continued regularly. However when the values are not visible, the position of the box does not need to be sampled. Thus it can be skipped and the position of the ray can be advanced up to the exit intersection point of the current box. Based on this exit point, the next cell can be determined which is again checked for its min/max values. This process is continued until the ray reaches the end of the volume or the ray accumulates to full opacity.

Calculation of Entry and Exit Points

The calculation of entry and exit points is done using a *slab-based* approach. A slab is defined as the space in between two parallel planes [TK86]. Since a box can be constructed from six parallel planes, its inner space is made up of three slabs. The intersection of a ray and a box can be described as the intersection of a ray with a set of slabs. Based on the origin and the direction of the ray, intersections with a box are calculated per slab. The concept is illustrated in Figure 7.7. Slabs are tested pairwise for an intersection with a ray. Starting at the origin of the ray, the first and the last intersection points between the ray and two slabs are calculated. Near and far intersection points are stored and the next pair

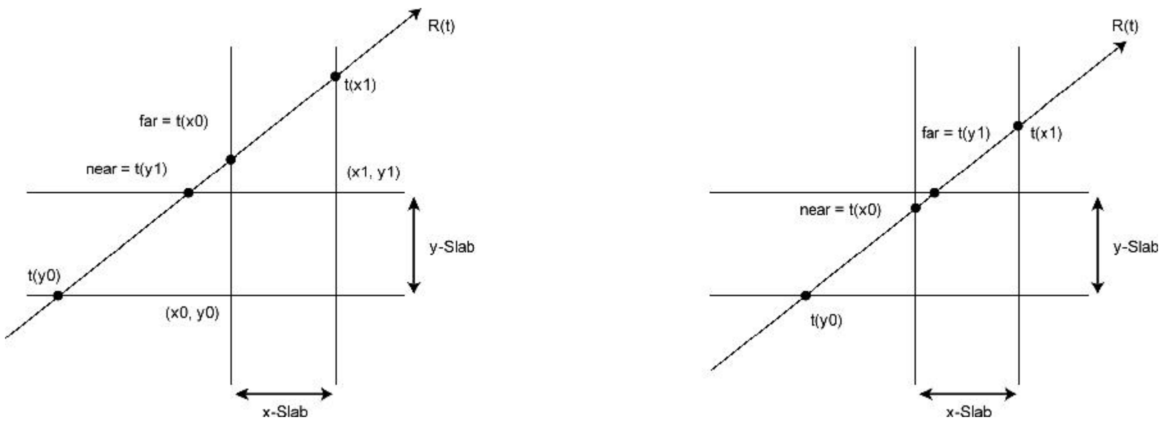


Figure 7.7: Raybox intersections are calculated based on slabs [TK86].

of slabs is tested. The ray misses the box if after all tests the largest hit point with the near slab is greater than the smallest hit point with the far slab. Otherwise the ray intersects the box.

ESP based on a min/max octree requires a recomputation of the octree after each change of the transfer function; therefore it is appropriate to only allow relatively shallow trees. In our tests we experimented with maximal tree depths of 5–6, which could be calculated quickly without disturbing the interactivity of the application too much. Implementing this feature is currently ongoing. It is mostly complete but was not integrated into the existing editor application. This is something that needs to be completed in a future version.

Chapter 8

Evaluation

In this chapter the current results of applying the occlusion spectrum to seismic datasets will be discussed. These include artificially generated datasets as well as sections from real-world data sets. Another new approach of this work is the use of different seismic attributes than the original amplitudes for calculating occlusion. The current results of this will be described in this chapter. Furthermore the new sampling algorithm will be analyzed with respect to its execution speed and different implementation will be compared to each other.

8.1 Classification Results

In order to evaluate the concept of occlusion-based classification several different datasets have been used as classification targets. These include artificial datasets as well as examples from real-word surveys that have been provided by the VRGeo members. Data in a seismic dataset is normally distributed with a high proportion of noise. To achieve realistic test scenarios, datasets were generated that consisted of Gaussian noise with different embedded structures. The structures themselves were either completely solid, gradient-based or consisted also of noise. The intensity ranges for the embedded structures were positioned close to the middle peak. This was done because structures that are based on velocity anomalies are generally easy to classify by an opacity mapping. Amplitudes around the middle peak, which are usually caused by noise, can thus be removed by setting them completely transparent so only the remaining minimum and maximum amplitudes are displayed.

8.1.1 Occlusion Based on Seismic Amplitudes

Channel Systems as Classification Target

There are several subsurface structures that are of interest in the context of hydrocarbon exploration. Besides the stratigraphic features such as faults and unconformities (see chapter 2), one integral part in the exploration for oil and gas is *channel detection*. Channels can sometimes be difficult to identify because of structural complexity in their neighborhood. However they have certain unique characteristics that distinguish them from other geological features. First and foremost channels are elongated structures. In geological terms channels are therefore known as *meanders*, which refers to their sinuous shape that all subsurface channels have in common [MC08]. However the most common geological features in seismic data are planar and laterally extended so they are different in shape.

Artificial Example

A first classification result is shown in Figure 8.1. The dataset shown consists of an artificial channel arm that is embedded into Gaussian background noise. The shape of the channel system is based on a sinus function and aims at copying the meandering shape of real-world channel systems. The mapping that is used in this example was determined by our adaptive selection method (Chapter 6). The channel can be seen as a distinct peak inside the spectrum. The image series in Figure 8.1 demonstrates the difference between one-dimensional opacity-based classification and two-dimensional occlusion-based classification. The effect of an opacity mapping can be achieved using the occlusion spectrum by defining a selection area that spans over the whole range of available occlusion values for a given intensity range. This gives the exact same result as an opacity mapping that renders all voxels transparent, that are located outside of the given intensity range. Even though the general shape of the channel can be seen in the first image, there is still very much noise present that shares the same intensity values as the channel. By moving the selection polygon upwards more and more noise is canceled, since only the structure itself generates high occlusion values. By finally selecting only the most upper occlusion values, which make up the distinct peak of the channel, most of the noise is canceled and the structure becomes visible clearly. It must be noted that all of the images are done within the same intensity range, whereas the amount of the occlusion range was reduced stepwise.

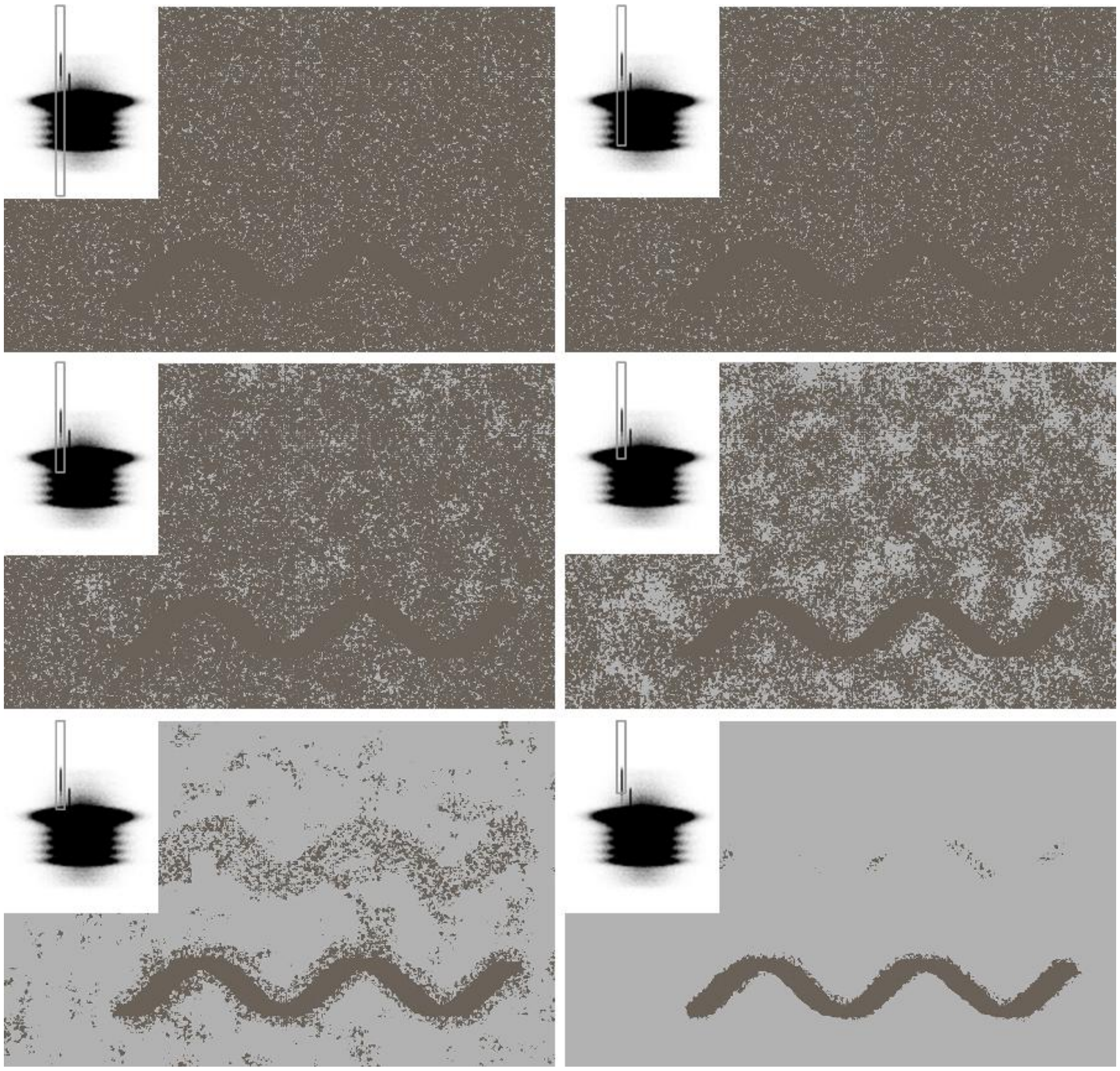


Figure 8.1: The channel shows as a distinct peak inside the spectrum. By selecting only high occlusion values, more and more noise can be removed from the visualization.

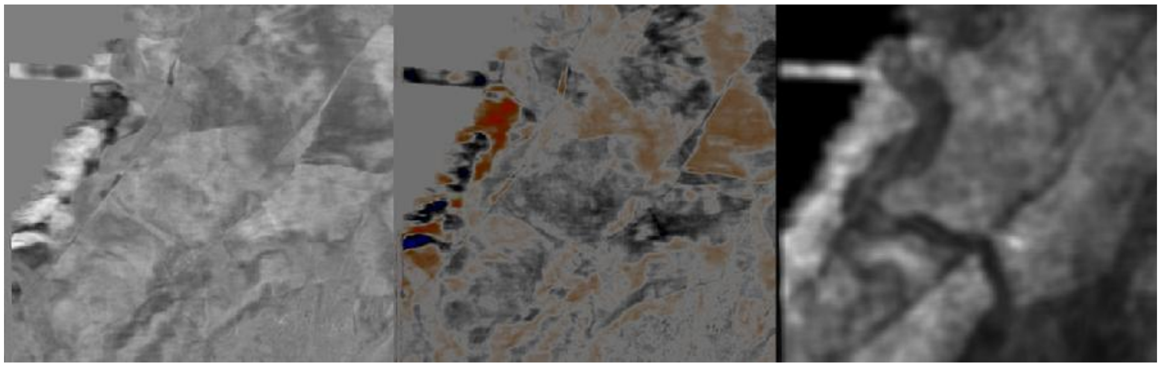


Figure 8.2: Seismic timeslice. Plain signal amplitude (left), rendered with a basic color table (middle) and rendered as slice through the occlusion volume (right).

Real-World Example

The next tests are done based on original seismic amplitudes as this is the form that all seismic datasets are available in. If not stated otherwise, all remaining screenshots of seismic datasets are taken from the Parihaka 3D survey that was recorded in New Zealand. Figure 8.2 shows a timeslice through a channel system. The display of the plain signal amplitudes can be enhanced by applying a color table. However it is still difficult to recognize the shape of the channel arm. On the very right side the same section of the dataset is rendered but this time the slice is not taken from the original seismic volume, but from the precomputed occlusion volume instead. In this view the path of the channel can be seen more clearly. However it must be noted that the occlusion will always be a bit blurry, which is due to the averaging that is performed and finer details will get lost. This result indicates that seismic occlusion is probably more suitable for detecting large-scale structures than small and more subtle features.

Another clear example of a subsurface channel system can be seen in Figure 8.3, which is taken from the Parihaka survey. The image shows a series of timeslices. The channel can be seen directly by looking at the original amplitudes, however local contrast is bad and its exact shape and path are unclear. Figure 8.4 shows the exact same series of timeslices, but this time not taken from the seismic volume, but from the occlusion volume instead. By choosing an appropriate visibility function, contrast between channel and other surrounding features can be enhanced leading, to a much clearer visualization.

In addition to the slice-based visualization, the same region of this dataset was also rendered volumetrically in terms of occlusion, which is shown in TODO. In this test again

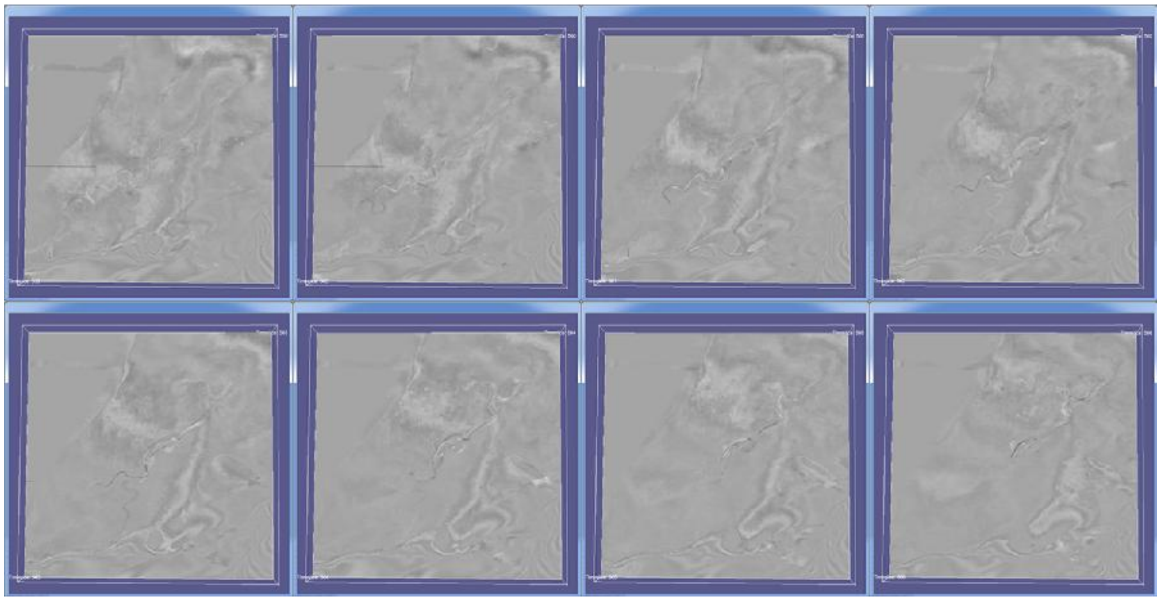


Figure 8.3: Series of time slices through the Parihaka dataset showing evidence of a meandering channel system. Only plain seismic amplitudes are displayed.

occlusion was generated based on a visibility mapping. This example serves as a good example to show the difference between regular opacity mapping and occlusion-based classification. By selecting the whole range as seen in the left picture, the results of an opacity mapping can be simulated. The path and shape of the channel can not be seen. By selecting only regions with higher occlusion values some of the surrounding noise can be removed and the channel becomes visibly clearer.

8.1.2 Occlusion Based on Seismic Sweetness

Channel detection is frequently done by using seismic sweetness (see chapter 2), which is a complex attribute that combines reflection strength and instantaneous frequency. Since the original paper only describes occlusion based on signal strength it was investigated if others seismic attributes might also be good candidates for calculating occlusion. It was suggested by one of the VRGeo members to use seismic sweetness. Therefore a direct comparison was performed. In this case the renderings were done in OpendTect since implementing support for different seismic attributes within the editor application would have been beyond the scope of this thesis. The data for this test was therefore generated as follows. First the desired subslice was extracted from the main survey and stored as a separate volume. The

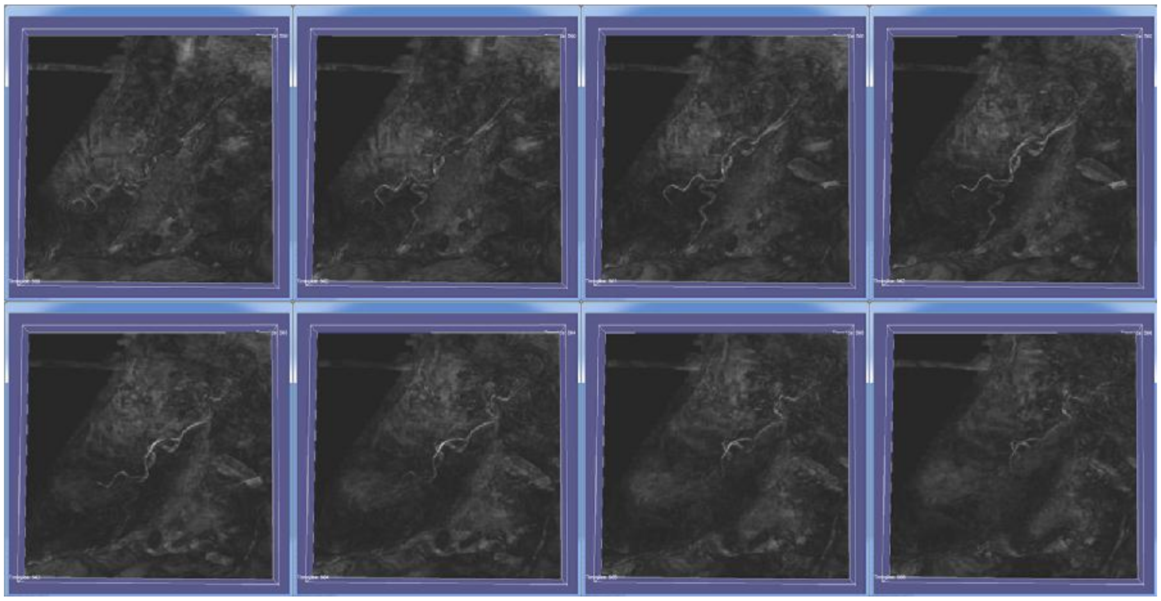


Figure 8.4: Same area as shown in Figure 8.3. This time rendered in terms of occlusion.

volume was imported into the demo application and occlusion was generated as described earlier. Since the application supports exporting into the SEG-Y format, the results could be integrated into OpendTect. Occlusion and sweetness could then be directly compared. The results of this first test are shown in Figure 8.5. On the left side is the regular occlusion (left bottom) based on original amplitudes (left top), whereas on the right side there is occlusion, which in this case is based on seismic sweetness (right top). At first sight both results look very similar and the result was therefore rated as inconclusive.

Direct Comparison of Occlusion and Sweetness

In order to better understand how this result originated, another test was performed. This time regular occlusion was directly compared to seismic sweetness. The results of this second test are shown in Figure 8.6. The two timeslices again look extremely similar, which leads to the question how these two attributes are related to each other. One notable similarity is the fact that both negative as well as positive amplitudes are displayed similarly. And this is due to the fact that seismic sweetness is based on the so-called *energy envelope*, which is defined as,

$$a(t) = \sqrt{x^2(t) + y^2(t)}, \tag{8.1}$$

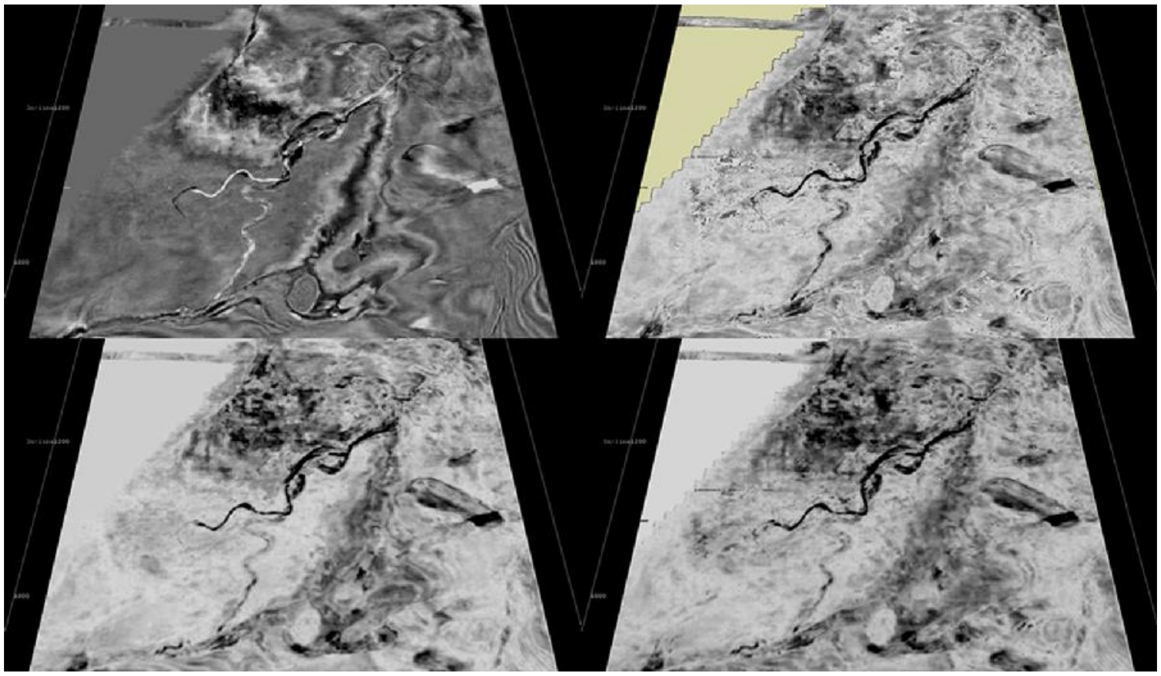


Figure 8.5: Original seismic amplitudes (top left), occlusion based on seismic amplitudes (bottom left), seismic sweetness (top right) and occlusion based on seismic sweetness (bottom right).

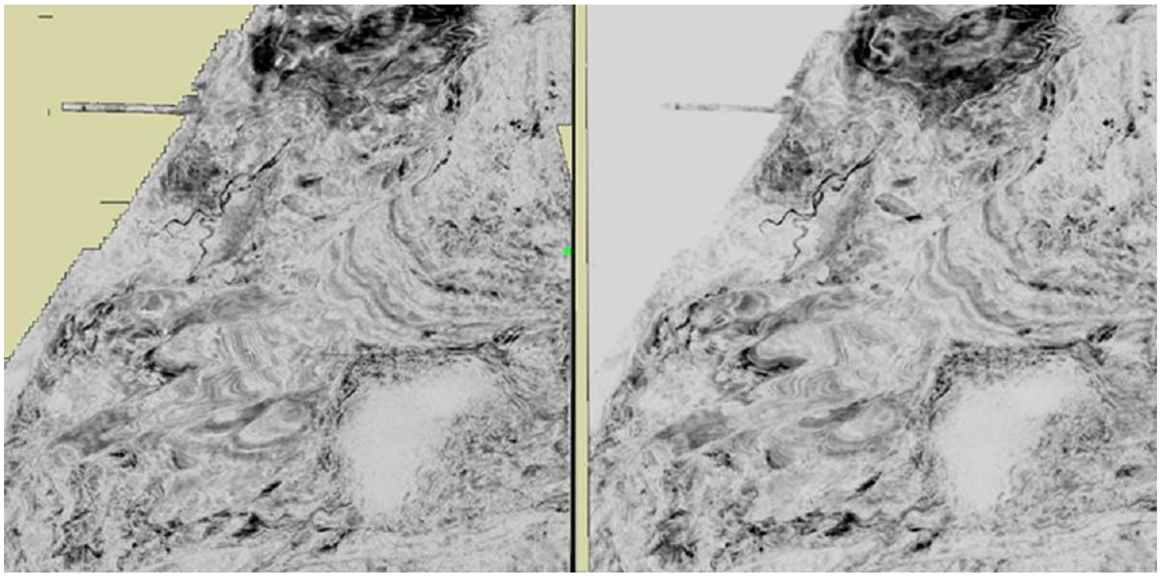


Figure 8.6: Direct comparison between seismic occlusion (left) and seismic sweetness (right).

where x and y are the real and imaginary part of the complex seismic trace respectively. This term will always be positive and by looking at the helix of the complex seismic trace it becomes obvious that negative and positive amplitudes are treated equally. This explains the optical appearance of the right screen shot in Figure 8.6. In a similar fashion a symmetric visibility mapping also combines positive and negative amplitudes, and generated occlusion values will therefore also always be positive. At this point it must be noted that this is only true for symmetrical mapping functions; the results of linear mappings for example will differ.

Even though these two attributes have very similar optical characteristics, there are differences. First of all the overall picture of seismic sweetness will be more sharp and contain more subtle details, because sweetness is an instantaneous attribute, that is calculated on a single voxel basis whereas occlusion is a spatial attribute that takes adjacent voxels into consideration. This leads to the blurry appearance of the occlusion. But this averaging also leads to a reduction in noise due to the averaging that is performed (Figure 8.7).

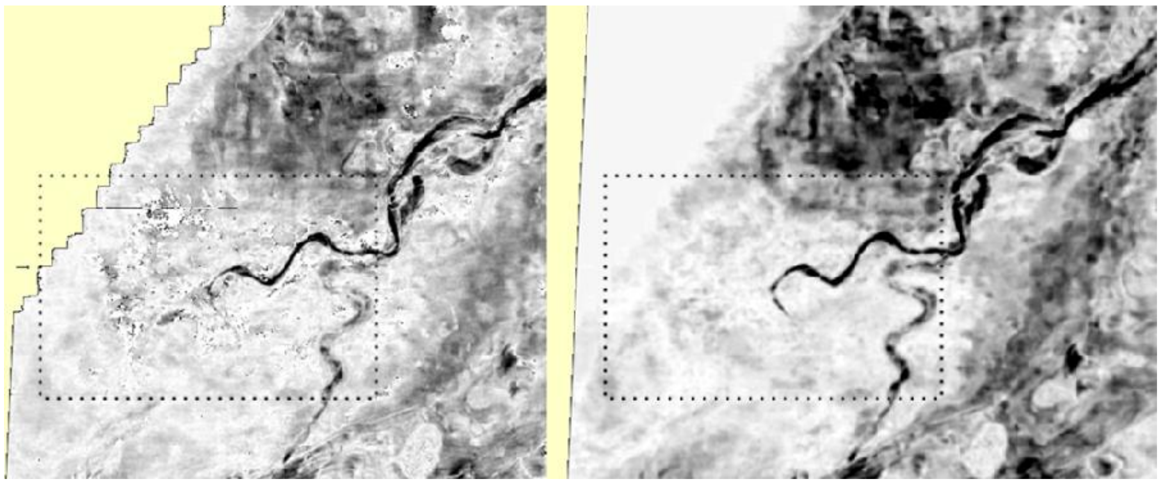


Figure 8.7: Sweetness (left) is more detailed, whereas occlusion (right) seems a bit blurry. However occlusion is less noisy due to averaging that is performed during generation.

8.2 Performance of Sampling Algorithm

One integral part of this work is a novel algorithm for precomputing occlusion and the modified version for generating local histograms. In its basic form occlusion generation has a complexity of $O(n^3)$. Since it was necessary to quickly experiment with different parameters, a new approach was needed. Execution times of the very first algorithm could easily reach multiple minutes or even hours and this was unacceptable. The new algorithm only considers border voxels and therefore builds occlusion as a difference between the neighborhood of adjacent voxels; the complexity could be reduced effectively to $O(n^2)$. Both algorithms have been implemented in a sequential and a parallel version which are executed on the CPU and the GPU respectively. In order to get a reasonable comparison between brute-force and border-based occlusion both versions have been additionally implemented for execution on the GPU.

In this section the original algorithm will be compared to the new one. One subject of this comparison is of course the difference in execution speed between the different versions. In addition to that a comparison will be done regarding the number of texture samples that are necessary for building the average of the neighborhood. In order to perform different tests, a couple of artificial datasets have been generated. These are cubic volume datasets, that are filled with a constant intensity. There are five different datasets with dimensions stretching from 50^3 up to 250^3.

Size	Radius	CPU / BF	CPU / Cached	Samples / BF	Samples / Cached
50	2	0,0250886	0,0476705	3726352	3803184
50	4	0,0848835	0,0617294	13713408	42909696
50	6	0,233118	0,0909095	28310544	151095600
50	8	0,478245	0,123935	46044160	352644096
50	10	0,832277	0,174792	65610000	663390000

Table 8.1: Comparison between brute-force and cached sampling implementation. Given volume dimension was 50^3.

Size	Radius	CPU / BF	CPU / Cached	Samples / BF	Samples / Cached
250	2	3,22055	5,59155	988047936	493031952
250	4	11,6239	7,80657	7809531904	1940574208
250	6	32,2356	10,9487	26039617344	4296009744
250	8	71,4034	15,8192	60976889856	7513666560
250	10	135,211	23,5962	1,17649E+11	11548810000

Table 8.2: Comparison between brute-force and cached sampling implementation. Given volume dimension was 250^3.

8.2.1 Brute Force vs. Cached

The first comparison has been made between two sequential implementations of the original brute-force algorithm and the cached sampling algorithm. Subject of comparison are the execution speed and the number of samples that are taken from the original volume texture. As an example two results from a first test run are shown in Table 8.1 and Table 8.2. The remaining measurements can be found in Figure 8.11.

It can be seen that the execution times as well as the number of samples for the brute-force (CPU / BF) implementation grow in a cubic way. This was expected as the complexity of this approach is $O(n^3)$. Compared to that the cached (CPU / Cached) version shows quadratic behavior in respect to execution times and number of samples. This is due to the reduction in complexity of this approach. Using the brute-force algorithm, the radius directly affects the sampling area in three dimension, however the new cached version is only affected in two dimensions. An increasing radius will only lead to a growth of the two border planes.

In Figure 8.8 the results from Table 8.1 and Table 8.2 are plotted in terms of execution time in milliseconds against radius. Even though the difference in execution time can be considered minor with small volume dimensions (around 0.65 ms), it becomes more obvious

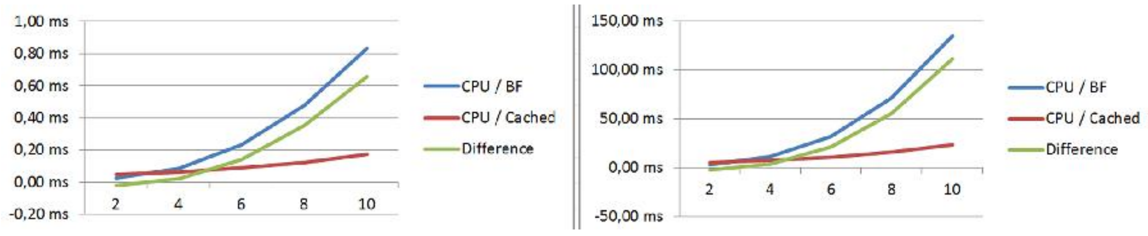


Figure 8.8: Execution times of brute-force and cached CPU implementation.

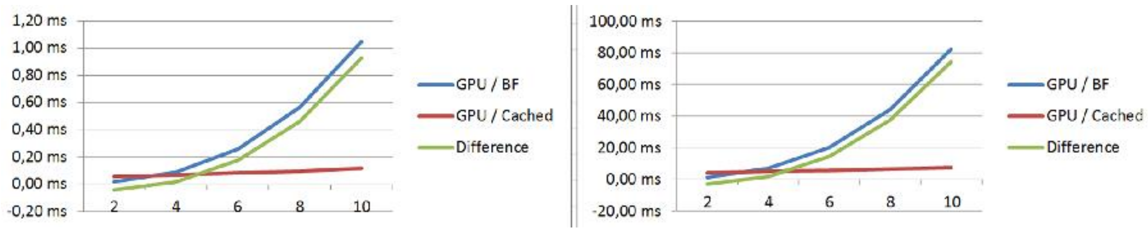


Figure 8.9: Execution times of brute-force and cached GPU implementation.

with increasing dimensions (around 111.61 ms).

The second comparison has been made between the GPU implementations of both approaches. The results are shown in Table 8.3 and Table 8.4. Overall the results are similar to the first test. However overall execution times are lower compared to the CPU implementations. This is probably due to the difference in memory bandwidth which, is higher on the GPU. A direct comparison between both CPU-based and both GPU-based implementations is shown in Figure 8.10. It can be seen that the GPU-based brute-force algorithm is actually slower than the CPU version as long as the radius of the volume is small. In such cases a basic implementation will be faster on the CPU. The reason for this is that even though the radius of the sampling area is relatively small, the whole volume needs to be transferred to the GPU memory. After the volume has been sampled, the data needs

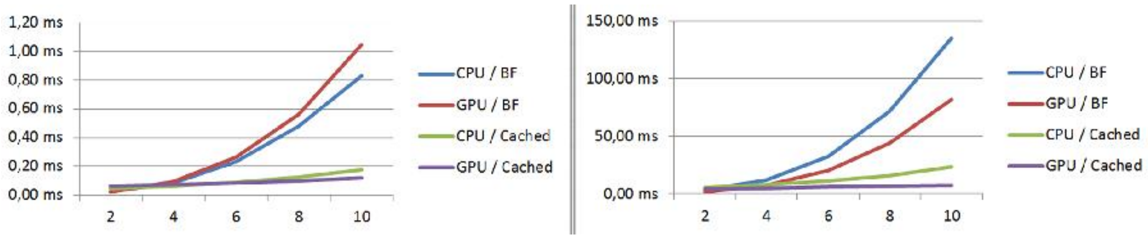


Figure 8.10: Comparison of execution times between CPU and GPU implementation.

Size	Radius	GPU / BF	GPU / Cached	Samples / BF	Samples / Cached
50	2	0,0229893	0,0612501	7530	3726
50	4	0,0930714	0,0719725	56623	13713
50	6	0,261664	0,0855891	179406	28311
50	8	0,564266	0,0992657	398688	46044
50	10	1,04653	0,118405	729000	65610

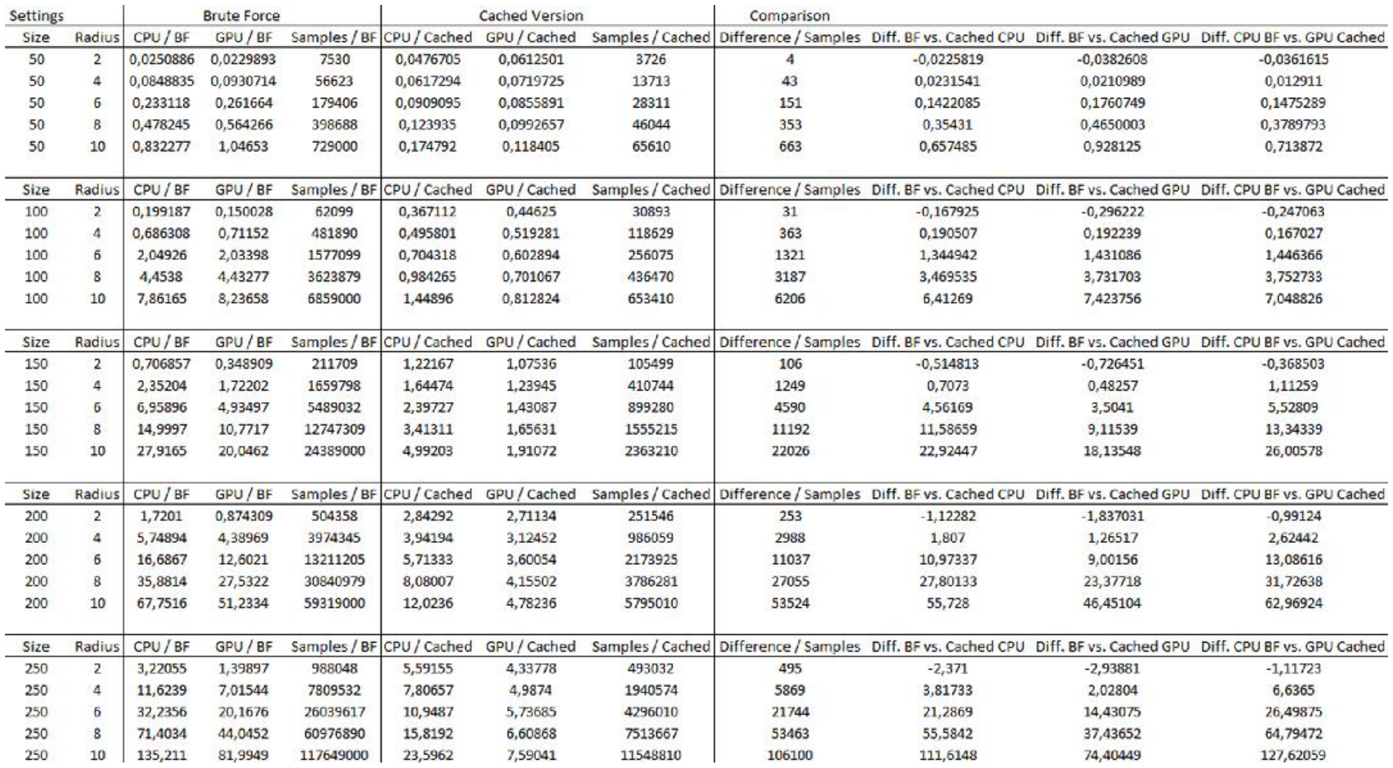
Table 8.3: Comparison between brute force and cached implementation with volume dimensions of 50^3.

Size	Radius	GPU / BF	GPU / Cached	Samples / BF	Samples / Cached
250	2	1,39897	4,33778	988048	493032
250	4	7,01544	4,9874	7809532	1940574
250	6	20,1676	5,73685	26039617	4296010
250	8	44,0452	6,60868	60976890	7513667
250	10	81,9949	7,59041	117649000	11548810

Table 8.4: Comparison between brute force and cached implementation with volume dimensions of 50^3.

to be transferred back to the host system. This overhead however is minor in relation to the number of samples that are taken from the volume texture with increasing dimensions. At volume dimensions of 250^3 the GPU-based brute-force version needs around 81.9 ms to execute whereas the CPU-based version needs 135.2 ms. This makes the GPU version around 1.65 times faster. This factor is very likely to increase further with bigger volume dimensions as can be suspected from the path of the two figures. The difference between the two cached versions is even bigger with around 23.5 ms for the CPU and 7.5 ms for the GPU version. In this case the GPU version is around 3.1 times faster.

Processing a volume with dimensions 250^3 and radius 10 takes around 135 ms on the CPU in a brute-force way. The same volume can be processed with the cached GPU version in around 7.5 ms. Therefore the cached GPU version is around 17.8 times faster than the brute-force CPU version. This difference is however very likely to further grow with increasing sampling areas.



Settings		Brute Force			Cached Version			Comparison			
Size	Radius	CPU / BF	GPU / BF	Samples / BF	CPU / Cached	GPU / Cached	Samples / Cached	Difference / Samples	Diff. BF vs. Cached CPU	Diff. BF vs. Cached GPU	Diff. CPU BF vs. GPU Cached
50	2	0,0250886	0,0229893	7530	0,0476705	0,0612501	3726	4	-0,0225819	-0,0382608	-0,0361615
50	4	0,0848835	0,0930714	56623	0,0617294	0,0719725	13713	43	0,0231541	0,0210989	0,012911
50	6	0,233118	0,261664	179406	0,0909095	0,0855891	28311	151	0,1422085	0,1760749	0,1475289
50	8	0,478245	0,564266	398688	0,123935	0,0992657	46044	353	0,35431	0,4650003	0,3789793
50	10	0,832277	1,04653	729000	0,174792	0,118405	65610	663	0,657485	0,928125	0,713872
Size	Radius	CPU / BF	GPU / BF	Samples / BF	CPU / Cached	GPU / Cached	Samples / Cached	Difference / Samples	Diff. BF vs. Cached CPU	Diff. BF vs. Cached GPU	Diff. CPU BF vs. GPU Cached
100	2	0,199187	0,150028	62099	0,367112	0,44625	30893	31	-0,167925	-0,296222	-0,247063
100	4	0,686308	0,71152	481890	0,495801	0,519281	118629	363	0,190507	0,192239	0,167027
100	6	2,04926	2,03398	1577099	0,704318	0,602894	256075	1321	1,344942	1,431086	1,446366
100	8	4,4538	4,43277	3623879	0,984265	0,701067	436470	3187	3,469535	3,731703	3,752733
100	10	7,86165	8,23658	6859000	1,44896	0,812824	653410	6206	6,41269	7,423756	7,048826
Size	Radius	CPU / BF	GPU / BF	Samples / BF	CPU / Cached	GPU / Cached	Samples / Cached	Difference / Samples	Diff. BF vs. Cached CPU	Diff. BF vs. Cached GPU	Diff. CPU BF vs. GPU Cached
150	2	0,706857	0,348909	211709	1,22167	1,07536	105499	106	-0,514813	-0,726451	-0,368503
150	4	2,35204	1,72202	1659798	1,64474	1,23945	410744	1249	0,7073	0,48257	1,11259
150	6	6,95896	4,93497	5489032	2,39727	1,43087	899280	4590	4,56169	3,5041	5,52809
150	8	14,9997	10,7717	12747309	3,41311	1,65631	1555215	11192	11,58659	9,11539	13,34339
150	10	27,9165	20,0462	24389000	4,99203	1,91072	2363210	22026	22,92447	18,13548	26,00578
Size	Radius	CPU / BF	GPU / BF	Samples / BF	CPU / Cached	GPU / Cached	Samples / Cached	Difference / Samples	Diff. BF vs. Cached CPU	Diff. BF vs. Cached GPU	Diff. CPU BF vs. GPU Cached
200	2	1,7201	0,874309	504358	2,84292	2,71134	251546	253	-1,12282	-1,837031	-0,99124
200	4	5,74894	4,38969	3974345	3,94194	3,12452	986059	2988	1,807	1,26517	2,62442
200	6	16,6867	12,6021	13211205	5,71333	3,60054	2173925	11037	10,97337	9,00156	13,08616
200	8	35,8814	27,5322	30840979	8,08007	4,15502	3786281	27055	27,80133	23,37718	31,72638
200	10	67,7516	51,2334	59319000	12,0236	4,78236	5795010	53524	55,728	46,45104	62,96924
Size	Radius	CPU / BF	GPU / BF	Samples / BF	CPU / Cached	GPU / Cached	Samples / Cached	Difference / Samples	Diff. BF vs. Cached CPU	Diff. BF vs. Cached GPU	Diff. CPU BF vs. GPU Cached
250	2	3,22055	1,39897	988048	5,59155	4,33778	493032	495	-2,371	-2,93881	-1,11723
250	4	11,6239	7,01544	7809532	7,80657	4,9874	1940574	5869	3,81733	2,02804	6,6365
250	6	32,2356	20,1676	26039617	10,9487	5,73685	4296010	21744	21,2869	14,43075	26,49875
250	8	71,4034	44,0452	60976890	15,8192	6,60868	7513667	53463	55,5842	37,43652	64,79472
250	10	135,211	81,9949	117649000	23,5962	7,59041	11548810	106100	111,6148	74,40449	127,62059

Figure 8.11: Algorithm comparison between versions.

8.2.2 Reduction of Texture Memory Access

The total number of texture samples needed for calculating the occlusion volume for a cubic dataset can be written as:

$$N_{samples} = n_{blocks} * s_{blocks} * n_{iterations} * 2 \tag{8.2}$$

where n_{blocks} is the number of blocks generated with vw being the width of one slice and bw being the width of one single block defined as:

$$n_{blocks} = (\frac{vw}{bw} + 1)^2 \tag{8.3}$$

and s_{blocks} as the number of samples taken per block:

$$s_{blocks} = (2 * r + bw)^2 \tag{8.4}$$

and the number of iterations:

$$n_{iterations} = vd + r + 1 \tag{8.5}$$

Applied to the former example it can be seen that this new approach only requires a fraction of the texture samples compared to the brute-force technique. Volume dimensions were 512^3 with a sampling radius of 25 and a block size of 32^2.

$$N = (\frac{vw}{bw} + 1)^2 * (2 * r + bw)^2 * (vd + r + 1) * 2 \tag{8.6}$$

$$N = (\frac{512}{32} + 1)^2 * (2 * 25 + 32)^2 * (512 + 25 + 1) * 2 \tag{8.7}$$

$$N = (17)^2 * (82)^2 * (538) * 2 \tag{8.8}$$

$$N_{samples} = 2.090.921.936 \tag{8.9}$$

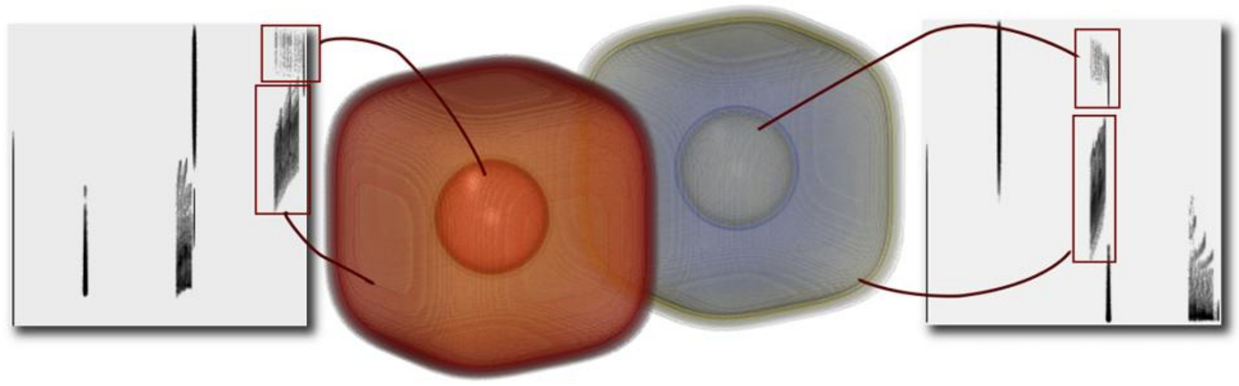


Figure 8.12: A mapping function that separates the structure on the left fails to separate the other structure [CK09].

8.3 Adaptive Mapping Selection

The visibility function defines how certain intensity ranges will be represented in the occlusion spectrum. Such a mapping function is generally implemented as a look-up table and can be understood as a separate transfer function. For each intensity value the mapping delivers exactly one occlusion value. Since this mapping function has a great impact on the appearance of the resulting spectrum it is necessary to compare different functions and compare the results regarding the contrast they deliver in the crossplot. Early tests indicated that different mapping functions highlight certain geological features better than others.

Consider the artificial datasets in Figure 8.12. Similar to the dataset shown in the beginning it consists of two nested structures, two spheres each of which surrounded by a cube. The structures differ in their intensity values and therefore one mapping that successfully leads to a separation of one structure in the occlusion spectrum fails to separate the other. Another mapping that separates the other structure leads to a bad visualization of the first structure. Hence it would be useful to be able to apply a mapping adaptively depending on the resulting contrast. A metric is needed in order to rate the quality of a mapping function. Since the goal of occlusion-based classification is a separation of structures inside the spectrum this metric should be derived from the crossplot itself. Firstly it must be defined what makes a good spectrum. A separation of structures is more likely to occur when the shape of the crossplot is vertically expanded, since a flat spectrum would be more and more like the original dataset.

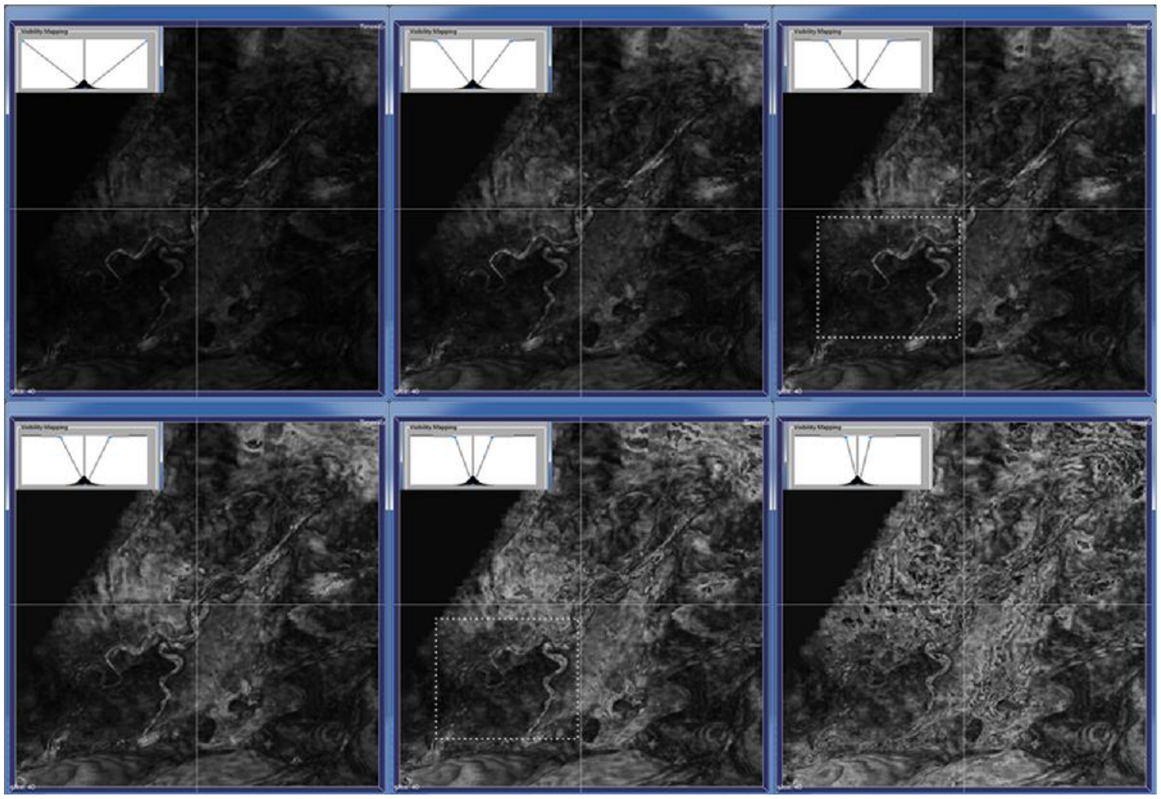


Figure 8.13: Certain mapping functions result in good contrast for some regions while others might require different mappings to gain a similar contrast.

Figure 8.13 shows the effect of different mapping functions applied to the channel system from the Parihaka survey. The upper arm of the channel is best visible using the third mapping function whereas the lower arm of the channel is best visible using the fifth. However contrast for the upper arm is bad and its shape can not be recognized clearly anymore. Therefore different mapping functions, which lead to the highest local contrast, must be used for different parts of the data.

Chapter 9

Conclusions and Future Work

The main focus of this work was to investigate if and how the occlusion spectrum, which is a classification technique from the medical domain, can be applied to volumetric data from the geoscientific domain. More precisely, it was investigated if it can be used for classification of volumetric seismic data, which is an integral part of modern hydrocarbon exploration.

In this regard it is necessary to have a basic understanding of the geological features and geophysical techniques that are used to acquire seismic datasets. These concepts were introduced in the beginning of the thesis. Afterwards the basic concepts of volume rendering were explained followed by a description of the volume rendering pipeline. Since the application further makes heavy use of Nvidias CUDA architecture for computing the occlusion, an introduction to GPU computing was provided.

The major bottleneck of the original version was the sampling algorithm for the generation of the occlusion information. It was done in a brute-force way and took depending on the parameters and the dimensions of the volume multiple hours to complete. To overcome this, a novel algorithm was developed that drastically reduces the number of texture samples needed for generating the weighted average. It is implemented in a parallel fashion and benefits from the enormous processing power of modern GPUs. This step was important because different settings and parameters needed to be tested within reasonable times. Changing a value and then waiting for multiple hours was unacceptable.

A demo application was developed that is used to preview and analyze the results of occlusion-based classification. It contains all necessary features such as volume data handling, preview rendering and multiple custom tools that are essential for a proper evaluation of the results. For example, the editor contains custom tools that aid the user in defining meaningfull visibility functions. For visual reference these can be interactively drawn over the amplitude histogram of the input data set. An interactive way to manipulate the visible section of the occlusion spectrum was offered in the form of a two-dimensional transfer function editor that contained several selection polygons. Moreover the results can be exported into common volumetric exchange formats and therefore be integrated into existing workflows for further comparison.

Several datasets have been used as classification targets. These include examples from real-word datasets as well as synthetic volumes. Promising results could be achieved regarding noise cancelation. It was shown that occlusion-based classification of seismic data sets is advantageous over basic opacity mapping, because it introduces an additional classification dimension that can be used to reduce the noisy sections in a seismic volume visualization.

Other results indicate that the concept is more suitable for detecting large-scale structures. Finer and subtle geological features may get lost, because of the averaging that is performed during occlusion generation. However this also leads to a less noisy visualization than compared to other seismic attributes such as sweetness.

In a future version, arbitrary oriented slices should be introduced, as they can be found in many common volume visualization toolkits such as Octreemizer [JMRB02] or OpendTect [dGB12].

The loss of detail information is one of the major problems during occlusion generation. The size and shape of structures will become more indistinct with increasing size of the sampling area. To overcome this, one approach is to perform the integration step with a bilateral edge-preserving filter. Instead of simply building the average of the neighborhood, the contribution of each voxel is controlled by its spatial distance but also depending on the difference in luminance. Such a bilateral filter will preserve sharp edges that define the shape and border of specific objects and structures.

In the current application it is possible to select voxels inside the volumetric data set and see the selection highlighted in the occlusion spectrum. However, currently only single voxels can be selected. This can be imporoved by letting the user select multiple voxels and then automatically building the selection polygons. For example, this can be done by building the convex hull for a set of selected voxels.

Occlusion has been compared to seismic sweetness and it was shown that both attributes are very similar in the way peaks and troughs are treated. As a consequence of the averaging, occlusion is less noisy but also contains less detailed information than sweetness. During the last VRGeo meeting several other attributes (semblance, coherency, relative impedance, cohesion) were suggested for comparison. One possible future investigation target is therefore how these attributes relate to occlusion and if they are more suitable for computing occlusion than the original seismic amplitudes.

Another interesting feature that was suggested by one of the VRGeo members, is the use of different shapes for the sampling mask, such as ellipsoids. Since seismic data consists of multiple layered strata, more planar sampling masks might improve the image quality.

To sum up, this work presented the current state of work regarding occlusion-based classification of seismic data; an approach that has not been done before. In this regard a new tiled algorithm was developed that is advantageous over the original brute-force approach because it is less complex and can be executed in a parallel fashion, making efficient use of advanced hardware features.

Bibliography

[Ame99a] American Geosciences Institute, *Agi learning modules, seismic attributes, first page knowledge base*, 1999.

[Ame99b] ______, *The hilbert transform*, 1999.

[AMM03] André Gerhardt, Marcos Machado, and Marcelo Gattass, *Two-dimensional opacity functions for improved volume rendering of seismic data*, 2003.

[And95] Andrew S. Glassner, *Principles of digital image synthesis (the morgan kaufmann series in computer graphics) 2 volume set*, Morgan Kaufmann, 1995.

[BCK11] Michael Bauer, Henry Cook, and Brucek Khailany, *Cudadma: optimizing gpu memory bandwidth via warp specialization*, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA), SC '11, ACM, 2011, pp. 12:1–12:11.

[Bro04] Alistair R. Brown, *Interpretation of three-dimensional seismic data*, 6 ed., AAPG, Tulsa, OK, 2004.

[Bru08] Bruce S. Hart, *Channel detection in 3-d seismic data using sweetness*, AAPG Bulletin, 2008.

[CK09] Carlos D. Correa and Kwan-Liu Ma, *The occlusion spectrum for volume visualization and classification*, IEEE Transactions on Visualization and Computer Graphics **15** (2009), 1465–1472.

[dGB12] dGB Earth Sciences, *Opendtect: Open source seismic interpretation system*, 2012.

[DW10] David B. Kirk and Wen-mei W. Hwu, *Programming massively parallel processors: A hands-on approach (applications of gpu computing series)*, Morgan Kaufmann, 2010.

[Env03] Inc Enviroscan, *Seismic reflection vs refraction*, 2003.

[FK03] Randima Fernando and Mark J. Kilgard, *The cg tutorial: The definitive guide to programmable real-time graphics*, Addison-Wesley, Boston, Mass. ;, London, 2003.

[G.F07] G.F. Moore, *Three-dimensional splay fault geometry and implications for tsunami generation*, Science, 2007.

[Gho12] Jayshree Ghorpade, *Gpgpu processing in cuda architecture*, Advanced Computing: An International Journal **3** (2012), no. 1, 105–120.

[Gor65] Gordon Moore, *Cramming more components onto integrated circuits*, Electronis Magazine, 1965.

[HN07] Pawan Harish and P. J. Narayanan, *Accelerating large graph algorithms on the gpu using cuda*, Proceedings of the 14th international conference on High performance computing (Berlin, Heidelberg), HiPC'07, Springer-Verlag, 2007, pp. 197–208.

[JMRB02] John Plate, Michael Tirtasana, Rhadames Carmona, and Bernd Fröhlich, *Octreemizer: a hierarchical approach for interactive roaming through very large volumes*, In Proceedings of the symposium on Data Visualisation 2002, 2002, p. 53.

[Kam10] Kamir Shahid, *Seismic interpretation basics*, 2010.

[KBH02] P. Kearey, M. Brooks, and Ian Hill, *An introduction to geophysical exploration*, 3rd ed. / ed., Blackwell Science, Malden, MA, 2002.

[KMJ$^+$06] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf, *Real-time volume graphics*, A K Peters, 2006.

[LBF05] Laurent Castanié, Bruno Lévy, and Fabien Bosquet, *Advances in seismic interpretation using new volume visualization techniques*, First Break Journal (2005).

[LC87] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, SIGGRAPH Comput. Graph **21** (1987), no. 4, 163–169.

[Low07] William 1939 Lowrie, *Fundamentals of geophysics*, Cambridge Univ Press, Cambridge, 2007.

[M. 01] M. Turhan Taner, *Seismic attributes*, CSEG - Canadian Society of Exploration Gepphysicists, Houston, 2001.

[MC08] J. Mathewson and Colorado School of Mines. Dept. of Geophysics, *Detection of channels in seismic images using the steerable pyramid*, Colorado School of Mines, 2008.

[MK07] Magnus Strengert Thomas Klein Thomas Ertl Martin Kraus, *Adaptive sampling in three dimensions for volume rendering on gpus*, 2007.

[MKG99] Lukas Mroz, Andreas König, and Meister Eduard Gröller, *Real-time maximum intensity projection*, 1999.

[MM04] Kenneth Dean Moreland and Kenneth Dean Morel, *Fast high accuracy volume rendering*, 2004.

[MRH10] Jörg Mensmann, Timo Ropinski, and Klaus H. Hinrichs, *An advanced volume raycasting technique using gpu stream processing*, GRAPP: International Conference on Computer Graphics Theory and Applications (Angers), INSTICC Press, 2010, pp. 190–198.

[Mur02] Murray Christie, *Thinking inside the box*, 2002.

[MWN01] Alan K. Faichney Michael W. Norris, *Seg y rev 1 data exchange format*, Society of Exploration Geophysicists, 2001.

[NVI09] NVIDIA Corporation, *Fermi: Nvidia's next generation cuda compute architecture*, 2009.

[NVI10a] NVIDIA Corporation (ed.), *Cuda c best practices guide*, vol. 2010, NVIDIA Corporation, Santa Clara, CA 95050, 2010.

[NVI10b] NVIDIA Corporation (ed.), *Nvidia cuda c programming guide*, vol. 2010, NVIDIA Corporation, Santa Clara, CA 95050, 2010.

[PBVG10] Daniel Patel, Stefan Bruckner, Ivan Viola, and Meister Eduard Gröller, *Seismic volume visualization for horizon extraction*, Proceedings of IEEE Pacific Visualization 2010, 2010, pp. 73–80.

[Pet09] Peter N. Glaskowsky, *Nvidia's fermi: The first complete gpu computing architecture*, 2009.

[PMM03] Pedro Mário Silva, Marcos Machado, and Marcelo Gattass, *3d seismic volume rendering*, 8 th Int. Congress of The Brazilian Geophysical Society (2003).

[Ric05] Rick Wilkinson, *Speaking oil and gas*, BHP Billiton Petroleum, 2005.

[Rit09] Rita Erfurt, *Hauptseminar im sommersemester 2007 medizinische bildverarbeitung*, Aachener Schriften zur Medizinischen Informatik, vol. V 2, B 4, Universitätsklinikum Aachen, Aachen, 2009.

[RL95] R. E. Sheriff and L. P. Geldart, *Exploration seismology*, Cambridge University Press, 1995.

[Rus10] Rusdinadar Sigit, *Post-stack seismic attribute*, Geo-Connection, 2010.

[TE02] S. Ellingsrud L. M. MacGregor T. Eidesmo, *Sea bed logging (sbl), a new method for remote and direct identification of hydrocarbon filled layers in deepwater areas*, 20 ed., First Break, 2002.

[TK86] James Kajiya Timothy Kay, *Ray tracing complex scenes*, SIGGRAPH, Pasadena, 1986.

[VRG12] VRGeo, *Vrgeo online portal: Virtual reality research for the geosciences*, 2012.

[Wol11] David Wolff, *Opengl 4.0 shanding language cookbook*, Packt Pub., Birmingham, UK, 2011.