NTNU
Det skapende universitet

GPU-Enabled Interactive Pore Detection for 3D Rock Visualization

Henrik Falch Hesland

Master i datateknikk
Oppgaven levert: Juni 2009
Hovedveileder: Anne Cathrine Elster, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

This project evaluates how Graphical Processing Units (GPUs) may be utilized to offload seismic computations in either multi-core and/or clustered environments. In particular, algorithms for analyzing seismic data will be parallelized using the GPU's processing power and taking advantage of the CUDA environment. Testing will be done on the HPC-LABs new NVIDIA Quadro FX 5800 and/or other appropriate systems.

Oppgaven gitt: 26. januar 2009
Hovedveileder: Anne Cathrine Elster, IDI

TDT4900
Complex Computer Systems,
Master Project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

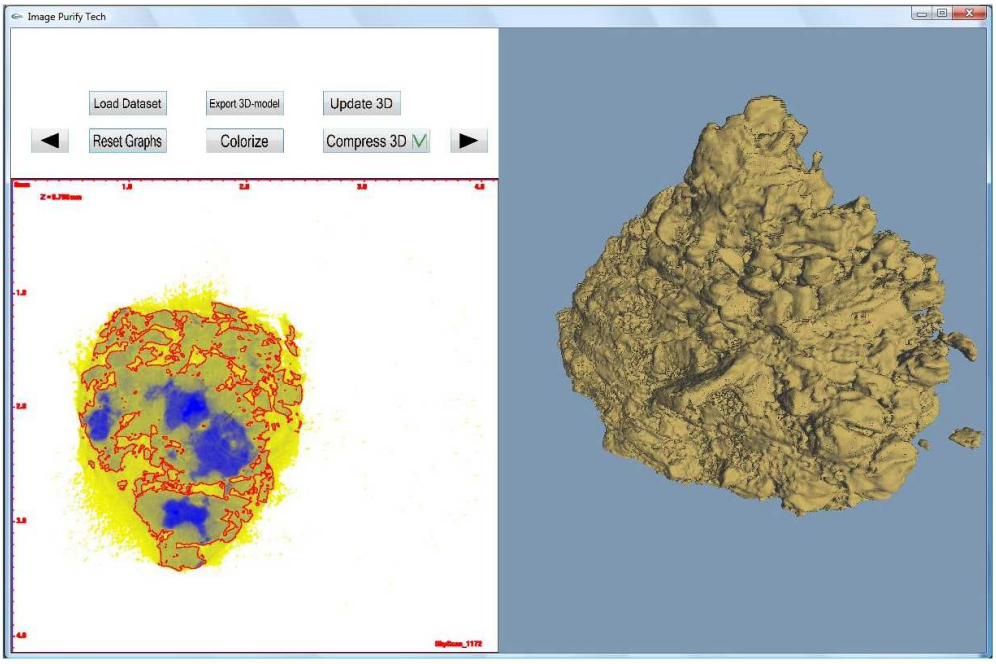
GPU-Enabled Interactive Pore Detection for 3D Rock Visualization

By: Henrik Hesland

Supervisor: Dr. Anne C. Elster

Co-supervisor: Thorvald Natvig

Trondheim, Norway, June 12, 2009



NTNU
Norwegian University of
Science and Technology

Abstract

Visualization of porous media is of great importance to several scientific fields, including the petroleum technology. The topic of this thesis arises from our collaborations with The Center for Integrated Operations in the Petroleum Industry. By being able to quickly analyze properties of porous rocks, they can get a better understanding of how to efficiently harvest oil since oil is typically held and stored within rock pores.

The petroleum industry typically uses Computed Tomography (CT) technology to scan rock samples for their internal structures. The resulting data is loaded into a computer program that generates 3D models of the rocks describing the 3D nature of its' internal structure. The scan data created from these scans will in most cases contain inaccuracies due to artifacts created while scanning.

In this thesis, we develop an application that interactively helps the user localizes the rock and pores in the CT scan data, allowing the user to create an image with a more accurate representation of the pores. We use digital image processing techniques to do an initial localization of the elements in the scan. The artifacts are then reduced by allowing the user to drag and pull on the line-data specifying the pores. Our implementation then uses this new representation to construct a 3D volume image that can be used in geophysical applications, like Schlumberger Petrel, for further analysis and simulation. The volume rendering part of our implementations builds directly on the authors project work with Eirik Ola Aksnes on GPU Techniques for Porous Rock Visualization completed last fall (2008).

Acknowledgements

The work on this Master Thesis has been carried out at the Department of Computer and Information Science in collaboration with the Center for Integrated Operations in the Petroleum Industry, at the Norwegian University of Science and Technology.

This Thesis would not have been possible without the support of several people. I wish particularly to express the gratitude to our supervisor Dr. Anne C. Elster who was incredibly helpful and offered invaluable assistance and guidance with her extensive understanding of the field. She has been a source of great inspiration, with her always encouraging attitude and generosity through providing resources needed for this project from her own income. I also wish to convey our deepest gratitude to PhD candidate Thorvald Natvig for his idea of the project; in addition without his technical knowledge and support this project would not have been successful. I wish to thank Schlumberger for providing us with Petrel and Ocean, and I wish to express our gratitude to Dr. Wolfgang Hochweller for helping us obtaining the Petrel and Ocean licenses. I wish to thank NVIDIA both for providing several GPUs to Dr. Anne C. Elster and her HPC-lab through her membership in the NVIDIA's Professor Partnership Program, and giving me the permission to use figures from NVIDIA Cuda Compute Unified Device Architecture Programming guide [10] in this thesis. Finally, I wish to thank Dark CodeX for giving me permission to use any screens from their 3D demos in this thesis [8].

Henrik Falch Hesland
Trondheim, Norway, June 13, 2009

Table of Contents

List of Figures

List of Tables

List of Abbreviations

NVIDIA CUDA	NVIDIA Compute Unified Device Architecture
OpenGL	Open Graphics Library
OpenCL	Open Compute Language
GLSL	OpenGL Shading Language
HLSL	High Level Shader Language
GPU	Graphics Processing Unit
GPGPU	General-Purpose computations on GPU
CPU	Central Processing Unit
UMA	Uniform Memory Access
NUMA	Non-Uniform Memory Access
MIMD	Multiple-Instructions Multiple-Data
SIMD	Single-Instruction Multiple-Data
SPMD	Single-Program Multiple-Data
ILP	Instruction Level Parallelism
SM#	Shader Model #
AMD CTM	AMD Close To the Metal
API	Application Programming Interface
SDK	Software Development Kit
CT	Computed Tomography
μCT	Microcomputed Tomography
VS	Vertex Shader
HS	Hull Shader
DS	Domain Shader
GS	Geometry Shader
PS	Pixel Shader
VBO	Vertex Buffer Object
PBO	Pixel Buffer Object
FBO	Frame Buffer Object

FPS	Frames Per Second
SGI	Silicon Graphics Inc
SM	Streaming Multiprocessor
SP	Stream Processor
SFU	Special Functional Unit
DRAM	Dynamic Random Access Memory
NTNU	Norwegian University of Science and Technology
MC	Marching Cubes
MT	Marching Tetrahedron
RGB	Red Green Blue (color channels)
RGBA	RGB Alpha (color channels)
DIP	Digital Image Processing
LoG	Laplacian of Gaussian
ASCII	American Standard Code for Information Interchange
MB	Mega Bytes
GB	Giga Bytes
GT	Regional Threshold
VT	Variable Threshold
VRT	Variable Regional Threshold
OS	Operating System

Chapter 1

Introduction

Traditionally, personal computers (PCs) used the central processing unit (CPU) for all general purpose computations. Most programs were executed serially, which made computations that required a lot of calculations – even if they all were the same – time consuming. Highly parallel problems like digital image processing, where each pixel in an image is manipulated, would often lead to millions of computations and several seconds of execution time. This made these algorithms unavailable for frame-based applications.

Today, 3D graphics accelerators have become powerful computational devices common in home computers. These accelerators, often called graphics processing units (GPU), are designed especially for computing the 3D graphics pipeline, which handles the different stages needed for translating a 3D representation to an image on the screen. 3D graphics is a highly parallelizable class of problems, which is handled by today's GPUs by computing hundreds of instructions simultaneously.

Since the GPU is such a powerful parallel computational device, it can not only be used for 3D graphics calculations, but also for general purpose programs which requires a high level of parallelism. A general purpose computation on a GPU (GPGPU) refers to applications traditionally computed on the CPU. GPGPU programming has the potential to give a huge speedup on suitable applications. However, it is more challenging to take advantage of the GPU this way since it requires parallel programming including efficient usage of several memory types.

One of the most popular GPGPU application programming interfaces (API) is NVIDIA CUDA (Compute Unified Device Architecture). NVIDIA CUDA has a syntacs similar to the C programming language and is a pro-

gramming model that focuses on low learning curve for developing applications that are scalable with the increase number of processor cores. To program NVIDIA CUDA efficiently, the programmer does not need to be familiar to the graphics pipeline, but needs to know the memory hierarchy of the NVIDIA Tesla architecture.

By the usage of GPGPU, several applications types get a significant speedup, including tasks such as fluid flow simulation, digital image processing and volume rendering.

1.1 Project Goals

It is important for the oil industry to analyze properties of rocks, so that a better understanding of conditions that affect oil production can be achieved [21]. With the use of non-destructive microcomputer tomography (CT scans) it is possible to make digital 3D representations of the internal pore structure of rocks. In order to get useful digital representations, detailed scans are needed, producing large volumetric data sets. Combined with real-time requirements for fast analysis and estimations of rock properties, there is an ever increasing demand for processing power.

When using a microcomputer tomography (μCT) scanner, the output will be a 3D representation of the densities of the rock and its pores. The representation will be slightly different from the real rock, called artifacts, because of both physical and scanner based inaccuracies. These artifacts will then also lead to inexact simulations of fluid or other calculations through the pores.

The main objective of this thesis is to find an improved algorithm for differentiating between the rock and the pores in a CT scan giving a density-representation of a core sample of the rock. The focus will be on providing the user with an interactive tool to enhance rock/pore detection.

In this thesis, we develop a visualization application that uses digital image processing requiring per-frame calculations to receive feedback. The types of digital image processing most suited for our approach is various types of segmentation techniques, where thresholding and edge detection is central. These techniques are highly parallelizable, so using the GPU for the computations is hence a natural choice. The user should also be able to use the segmented images to create a 3D volume of a CT scan, which is based

on previous work, explained further in Section 1.2.

1.2 Related Work

This thesis builds on this author's Master report done in collaboration with Eirik Ola Aksnes Fall 2008, entitled "GPU Techniques for Porous Rock Visualization" [1]. Some background material for this project is also added to this thesis. Our project [1], involved developing a GPU implementation of the Marching Cubes algorithm, for improved performance on large datasets. The algorithm takes a dataset of BMP-images as input, calculate the border-values between rock and pores, and use them to create a 3D volume. The border value is user-specified by a global integer value. In this thesis, we are going to improve the calculation of a border value, where the user will be able to observe the borders, and not think about actual values.

1.2.1 Related Implementations and Techniques

Image Processing and Artifact Removal

- [18, 36] contains detailed explanation of all the basic image processing techniques we have implemented, including basic thresholding, the Sobel Edge filter and Canny Edge detection. The books is used as curriculum in the subjects TDT4195 - Image Techniques and TDT4265 - Computer Vision at NTNU.

- [4] contains a good overview of the typical artifacts that can be generated by CT-scanning. It also contains short examples of how to avoid these artifacts. We did not use any of the techniques directly, but the explanation of why the artifacts appeared was a good overview.

- [37] explains a method to reduce non-linear artifacts in CT. It contains an approach to be done by the CT software vendors, and a similar method is probably used by the SkyScan software.

- [27] describes a way to threshold images by region based multi level thresholding, where the algorithm is able to find several regions in a color image. Can be used to solve similar types of problems, but is a different approach than we used.

Volume Rendering

- [16] contains the volume rendering method Marching Cubes, close to the previous implementation in [1]. It uses geometry shaders to compute the volume, where we are using NVIDIA CUDA.

- [33] is a PhD thesis using the volume rendering method Shear-Warp, which is a direct volume rendering method. It is a different approach than ours. Can in some cases reduce the memory used for the volume data, but requires per frame computations, instead of saving the triangles. The thesis also includes a good overview of different volume rendering methods.

- Algorithms which use different approaches to improve the Marching Cubes algorithm, in either performance or accuracy, and is potential future work: [41, 12]

1.3 Outline

This thesis is structured in the following manner:

In Chapter 2 - Parallel Computing and GPUs, we will explain some benefits of parallel computing and GPUs. In Section 2.1 we will explain some aspects of parallelism, and Section 2.2 will follow with an overview of what a GPU is. A history of the evolution of the GPU will follow, before explaining the graphics pipeline, GPGPU programming and graphics programming. Some parts of Sections 2.1, 2.2 and 2.3 are picked up from Eirik Ola Aksnes' and Henrik Hesland's paper GPU Techniques for Porous Rock Visualization [1].

In Chapter 3 - Porous Rock Structures and Volume Rendering we will first explain how porous rocks are related to the oil business in Section 3.1. Section 3.2 includes a short background on computer tomography (CT) scanning and an overview of the most important artifacts received from these scans. Then in Section 3.3 there is a quick overview of volume rendering techniques. Some of the parts of this chapter are also picked up from Eirik Ola Aksnes' and Henrik Hesland's paper GPU Techniques for Porous Rock Visualization [1].

In Chapter 4 - Digital Image Processing, a short overview of the field of digital image processing will be introduced. There will be found a short explanation of digital image processing basics in Sections 4.1 and 4.2, then following in Sections 4.3 and 4.4 a few more advanced techniques.

In Chapter 5 - Implementations, we will explain some thoughts around our implementation, the reasons of some of the algorithmic choices and some guidelines to follow when programming NVIDIA CUDA.

In Chapter 6 - Benchmarks and Results, we will first take a look at the performance of some different parts of our application. Then an evaluation of the visual results, where we compare the graphical output from several different threshold methods.

In Chapter 7 - Conclusions and Future Work, we will hold the conclusion of our work and a discussion of future work.

Chapter 2

Parallel Computing and GPUs

As stated by Eirik Ola Aksnes and Henrik Hesland in [1]: Moore's law predicts that the speed of computers would double about every two years. Today's computers have a high number of transistors inexpensively placed on an integrated circuit, but achieving this is getting harder. In fact, we have already hit the Power and Frequency Walls, which means that a processor cannot use a higher frequency without increasing the power. And increasing the power will also increase the heat generated when the aircooling is capped at 150W. Whatever the peak performance of today's processors, there will always be some problems that require or benefits from better processor speed. As explained in [2], there is a recent renaissance in parallel computing development. Due to the Power Wall, increasing clock frequency is not the primary method of improving processor performance anymore, parallelism is thou the future. Both modern GPUs and CPUs are concerned with the increasing power dissipation, and want to increase absolute performance but also improve efficiency through architectural improvements by means of parallelism.

Parallel computing often permit a larger problem or a more precise solution of a problem to be found within a practical time. Parallel computing is the concepts of breaking up a larger problem into smaller units of tasks that can be solved concurrently in parallel. However, problems often cannot be broken up perfectly into autonomous parts, so interactions are needed among the parts, both for data transfer and synchronization. The problem to be solved affects how easy it is to parallelize. If possible, there would be no interaction between the separate processes, each process requiring different data and productive results from its input data without need for result from other processes. However many problems are to be found in the middle, neither fully autonomous nor synchronized [44].

There are two basic types of parallel computers, if categorized based on their memory architecture [44]:

- Shared memory systems that have a single address space, which means that all processing elements can access the same global memory. It can be very hard to implement the hardware to achieve uniform memory access (UMA) by all the processors with a larger number of processors, and therefore many systems have non uniform memory access (NUMA).
- Distributed memory systems that are created by connecting computers together through an interconnection network, where each computer has its own local memory that cannot be accessed by the other processors. The access time to the local memory is usually faster than access time to the non-local memory.

Distributed memory will physically scale more easily than shared memory, as its memory is scalable with increase number of processors.

Today, parallelism have become the standard way of increase overall performance for both the CPU and GPU. Need appropriate forms of doing parallelism, which exploits the different architectures.

2.1 Forms of Parallelism

There are several forms of doing parallel computing. To frequently used are task parallelism and data parallelism.

Task parallelisms, also called *Multiple-Instruction Multiple-Data* (MIMD), focus on distribute separate tasks across different parallel computing nodes that operate on separate data streams in parallel. It can typically be difficult to find autonomously tasks in a program and therefore task parallelism can have limited scaling ability. The interaction between different tasks occurs through either message passing or shared memory regions. Communication through shared memory region poses the problem with maintaining memory cache coherency with increased number of cores, as most modern multicore CPUs use caches for memory latency hiding. Ordinary sequential execution of a single thread is deterministic, making it understandable. Task parallelism on the other hand even if the program is correct is not. Task parallelism is subject to faults such as race conditions and deadlock, as correct synchronization is difficult. Those faults are difficult to identify, which

can make development time overwhelming. Multicore CPUs are capable of running entirely independent threads of control, and are therefore great for task parallelism [25].

Data parallelism is a form of computation that implicitly has synchronization requirements. In a data parallel computation, the same operation is performed on different data elements concurrently. Data parallel programming is very convenient for two reasons. It is easy to program and it can scale easily to large problems. The *Single-Instruction Multiple-Data* (SIMD) is the simplest type of data parallelism. It operates by having the same instruction execute in parallel on different data elements concurrently. It is convenient from hardware standpoint since it gives an efficient hardware implementation, because it only needs to replicate the data path. However, it has difficulty of avoiding variable work load since it does not support efficient control flow. The SIMD models have been generalized to the *Single-Program Multiple-Data* (SPMD), which include some control flow. Making it possible to avoid and adjust work load if there are variable amounts of computation in different parts of a program [25], as illustrated in Figure 2.1.

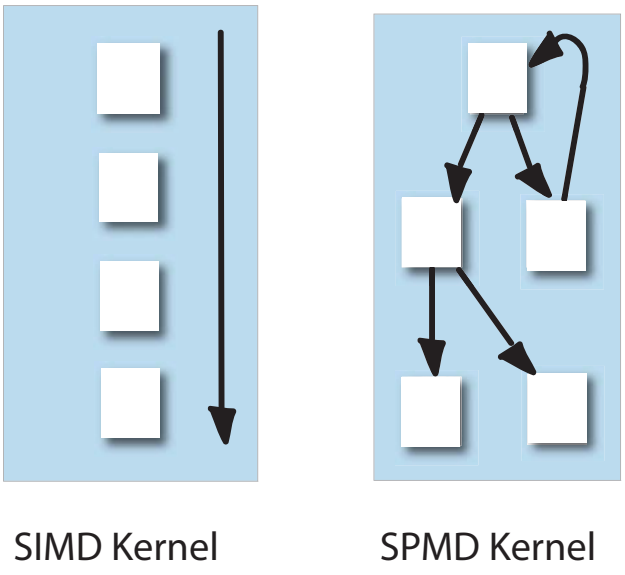


Figure 2.1: SPMD support control flow, SIMD does not.

Data parallelism is essential for the modern GPU as a parallel processor,

as they are optimized to carry out the same operations on a lot of data in parallel.

2.2 Graphics Processing Units

The Graphics Processing Unit (GPU) is a special-purpose processor dedicated for rendering computer graphics for a variety of tasks, ranging from three-dimensional (3D) visualization in games to drawing text in internet web browsers. Over the past 40 years, GPUs have made their way from research labs and flight simulators to commercial personal computers and other entertainment systems; a section including the evolution of the GPU follows in Section 2.2.1. The GPU's computational units and memory bandwidth has become quite computational powerful. The GPUs now is much more computational powerful than the CPUs on parallelizable calculations.

The CPU is designed to maximize the execution speed of a single thread of sequential instructions. It operates on different data types (floating-point and integers), performs random memory accesses and branching. Instruction level parallelism (ILP) allows the CPU to overlap execution of multiple instructions or even change the order in which the instructions are executed. The goal is to identify and take advantage of as much ILP as possible [42]. To increase performance the CPU uses much of its transistors to avoid memory latency with data caching, sophisticated flow control and to extract as much ILP as possible. There is a limited amount of parallelism that is possible to get out of a sequential stream of instructions, also known as the ILP Wall, and ILP causes a super linear increase in execution unit complexity and associated power consumption without linear speedup in application performance [24].

The GPU is dedicated for rendering computer graphics, and the primitives, pixel fragments and pixels can largely be processed independently and therefore in parallel (the fragment stage is typically the most computationally demanding stage [28]). The GPU differ from the CPU in the memory access pattern, as the memory access in the GPU are very coherent, when a pixel is read or write a few cycles later the neighboring pixel will be read or write. By organizing memory intelligently and hide memory access latency by doing calculations instead, there is no need for big data caches. The GPU are designed such that the same instruction operates on collections of data and therefore only need simple flow control. The GPU dedicate much more of its transistors for data processing than the CPU, as illustrated in Figure

2.2.

Figure 2.2: CPU and GPU transistor usage. Figure is taken with permission from [10].

The modern GPU is a mixture of programmable and fixed function units, allowing programmers to write vertex, fragment and geometry programs for more sophisticated surface shading, and lighting effects, this will be explained further in Section 2.2.2. The instruction set to the vertex and fragment programs has converged, in which all programmable units in the graphics pipeline share a single programmable hardware unit. To the unified shader architecture, where the programmable units share their time among vertex work, fragment work, and geometry work [28].

The GPU differentiate themselves from traditional CPU designs by prioritizing high-throughput processing of many parallel operations over the low-latency execution of a single thread. Quite often in scientific and multimedia applications there is a need to do the same operation on a lot of different data elements. GPUs support a massive number of threads, typically 61440 on a NVIDIA GeForce GTX 295, running concurrently and support the Single-Program Multiple-Data (SPMD) model to be able to suspend threads to hide the latency with uneven workloads in the programs. The combination of high performance, low-cost, and programmability has made the modern GPU attractive for applications traditionally executed by the CPU, for General-Purpose computation on GPUs (GPGPU). With the unified shader architecture, the GPGPU programmers can now target the programmable units directly, rather than split up task to different hardware units. To harvest the computational capability and at the same time allow

programmers to be productive, we need programming models that strike the right balance between low-level access to hardware resources for performance and high-level abstraction of programming languages for productivity.

2.2.1 The Evolution of GPUs

In the 1960s, the earliest applications with computer graphics were developed. In these years, the application types were mostly computer-aided design and flight simulators, followed shortly by entertainment applications, like computer games. The graphics processing were focused on controlling vector displays, which could represent objects by lines and wire frames [5]. These technologies lead to research of algorithms for projecting 3D-object representation on 2D display planes.

Later in the 70s, semiconductor memory was introduced, which gave a way to raster techniques using pixel representations of images, which lead the way from wire frame images to solid images. This got even further in the 80s, when the personal computer marked became commercial. IBM started producing 2D raster graphics cards, which met the demanding document processing applications. Then later in the 80s, hardware accelerations of fixed pipeline operations like geometry operations, with simplified polygonal representation, depth buffering, light models and some more [5].

3D acceleration add-in cards became available for personal computers in the early 90s. These graphics cards included functionality as texture mapping and a 3D graphics pipeline. The commercialization of the graphics cards lead to the standardization of graphics APIs, where OpenGL and Direct3D was the most commercially used, and is explained further in Section 2.4.

By the end of the 90s, the largest graphics card providers: NVIDIA and ATI produced cards with acceleration for functionality including fixed-function geometry processing, rasterization, texture-mapped fragment processing and depth-buffer pixel processing. The new graphics processors were started to be referred as graphics processing units (GPU), and became a major step in fields like medical imaging, visual simulation and the entertainment industry. The first programmable GPU was NVIDIA GeForce 3, released in 2001. To program this GPU, Microsoft Direct3D 8.0 with Shader Model 1.1 was the first commercial tool, which was based on small Assembly code-programs [29]. The graphics pipeline was also upgraded in the next few years, to Shader Model 3 (SM3), giving functionality to manipulate vertices and pixel fragments. To use these manipulation techniques, the user had

to program small manipulation programs called shaders, which will be explained further in Section 2.2.2 [5].

The introduction of the unified shader processors late in 2006, released with NVIDIA's graphics card GeForce 8800. NVIDIA GeForce 8800 enabled the possibility of Shader Model 4 (SM4), where the same graphics processors were used for both pixel and vertex shaders, and also the new geometry shader, explained further in Section 2.2.2. This also opened for new type of General Purpose GPU (GPGPU) programming. Earlier the GPGPU applications were programmed by using the vertex and pixel shaders, which then made it hard to program. The SM4 graphics cards opened up for GPGPU languages like NVIDIA CUDA and AMD CTM. The GPU then became fully programmable thanks to the new unified shader architecture, which also are using the same processors for all the programmable stages in the graphics pipeline. These new languages had a syntacs close to C/C++ and made it much easier and more effective to accelerate general purpose programs on the GPU.

In late 2008 the Shader Model 5 (SM5) were introduced in Microsoft DirectX 11, with its compatible hardware released in the middle of 2009. In this hardware, the hardware tessalators were introduced, with shaders added in the graphics pipeline for manipulating the tessellation hardware. Some details about the tessellation stage of the pipeline will follow in Section 2.2.2.

2.2.2 The Graphics Pipeline

An image on a screen can be synthesized from a 3D scene consisting a geometric shape and appearance descriptions, such as colors and textures for each object in the scene. In addition, there are environment descriptions such as lighting, atmosphere and etc. The result from the synthesizes is a 2D array, where each value represent a pixel, which can be shown on the screen. To synthesize the image, each object is rendered using the graphics pipeline. The new SM5 pipeline uses the step process shown in the Figure 2.3 [15]. Where the input to the pipeline is a representation of a 3D scene, and the output is either drawn directly on the monitor or written to a framebuffer. The SM4 programmable stages is marked as red, the fixed function parts of the pipeline blue and the features from SM5 green.

The graphics pipeline now contains 5 programmable shader-types. A shader is a kernel function, being computed in parallel on the GPU. A shader can manipulate the relevant data at a current part of the pipeline, a more

Figure 2.3: The graphics pipeline SM5.

detailed description of each of the shaders will be explained underneath. Manipulation of data, through shaders usually uses OpenGL's GLSL (OpenGL Shading Language) or DirectX's HLSL (High Level Shader Language) as the program language. They both are very similar to C/C++ syntacs.

Vertex Shader

A Vertex Shader (VS) is a special program with functionality for manipulating vertex data by using mathematical functions. The VS is taking a vertex as input, where the vertex contains information of position and color. The VS will then work on all the vertices in the scene independently and in parallel using one vertex per thread. A shader program can change the position of a vertex, or how it is seen (color), and sends the manipulated vertex further through the pipeline as the shaders output.

VS can be used to various types of different scene manipulations. Examples of this are: By changing positions of the vertices, there will be possible to simulate a fluid, like in Figure 2.4, which also includes some other shader effects. Another way to use the VS is simulating fog, by manipulating the color. Other types of practical usage are interpolation of primitives and transformation between coordinate-systems.

Figure 2.4: Vertex shaded water movement. The screen is taken with permission from [8].

Tessellation Process

Tessellation is the new feature for SM5, introduced in Microsoft Direct3D 11; it contains three parts: the Hull Shader (HS), the tessalators and the Domain Shader (DS), where the first and third part is fully programmable, while the tessellator stage is configurable.

After the VS are done with the vertex manipulation, the HS receives surface patches, where a HS is called for each patch. In the HS, first an optionally functionality is executed. This first functionality can convert control points from the input patch to another representation. An example of this is approximating Catmull-Clark subdivision surfaces with bicubic patches, which you can read more about in [31]. After the conversion, the new control points will be sent directly to the DS. The second functionality of the HS is the computation of the tessellation factors, which are conceded to the tessellation stage. There is a tessellation factor for each edge of the patch, where the factor decides how many pieces the edge will be split into.

The tessellator is a fixed-function stage. In the tessellation stage, it di-

vides the patches into multiple triangles or quads relative to the tessellation factor and the configurations of the tessalator. The output from the tessellator is the new vertices created, where they is passed directly into the DS.

The DS finally gets the vertices from the tessellator in patch parameterization coordinates. The DS operates on the vertices separately, where the functionality specializes on completion of the vertex data, before the vertices are sent further through the pipeline.

By the use of the tessellation process, there are several techniques that can be accelerated. They vary from rendering complex 2D curved shapes. The GPUs can render simple polygons, but when it comes to concave or self intersecting polygons, they need to be handled in a way to be able to give the right output to the display. Another technique which will be accelerated significantly with the tessellation is terrain rendering, where terrains can be represented geometrically, and by using Bezier surfaces transformed into triangles. The tessellation enables the use of sophisticated algorithms to evaluate level of refinement of terrain patches. To read about more examples of tessellation usage, take a look at [7].

Geometry Shader

As explained in [1], the Geometry Shader (GS) was introduced in SM4, which requires a GPU with unified shader support. The GS can be programmed to generate new graphics primitives, such as points, lines and triangles [29]. Input to the geometry shader is the primitives after they have been through the vertex shader and the tessellation process. When operating on the triangles, the GS's input will be the three vertices. For storage of the output, the geometry shader then uses a vertex buffer, or the data is sent directly further through the graphics pipeline, starting with the rasterization.

GS, are designed to efficient create geometry. An example of a geometry creation program is using the Marching Cubes algorithm, which can create a 3D model from a cloud of points, where the result can be as shown in Figure 2.5. Marching Cubes is explained further in [1].

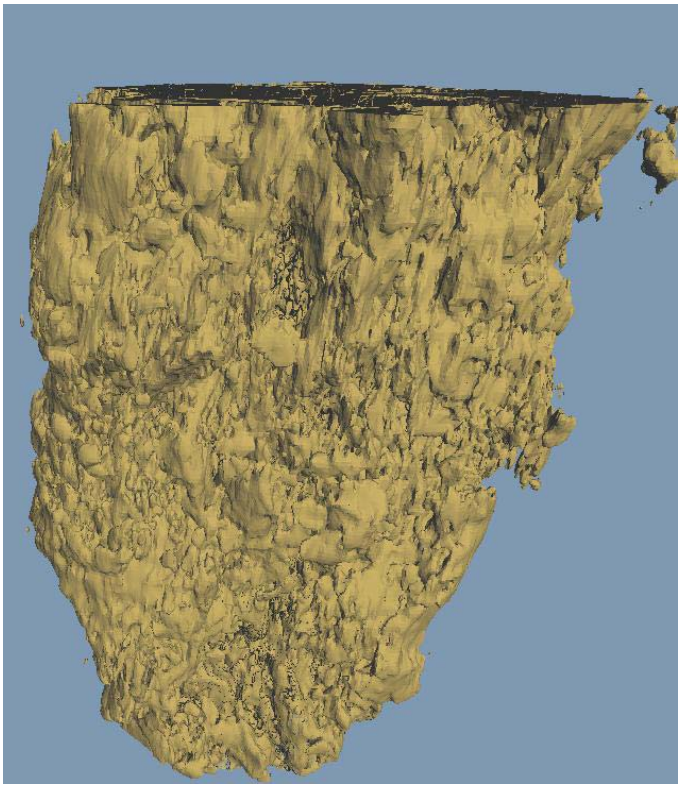


Figure 2.5: A complex 3D volume.

Rasterization

The rasterization part of the rendering pipeline has fixed-function hardware, meaning it is not programmable. This part of the pipeline basically transforms the 3D space into a 2D frame, by using 3 steps. The first step is the transformation of coordinate systems, placing all the vertices at the right places in the 3D world and transforms the objects to 2D screen coordinates by multiplying it to the transformation matrix. Then the next step is the clipping stage. The clipping is basically just fitting the triangles in the scene inside the frame, by clipping and removing information outside the screen. A common technique to do the clipping is Sutherland-Hodgeman, which is explained further in [20]. The final step of the rasterization is the scan-conversion, which is a method for filling the triangles that now are in 2D space. The scanline algorithm is the most common for this approach, deciding the inside and outsides of the triangles in the scene. For more information of the rasterization stage of the graphics pipeline, take a look at [20].

Pixel Shader

The Pixel Shader (PS) is used for manipulation of individual pixels, and allow the programmer to manipulate the part of the graphics pipeline concerning shading and texturing. The input to a PS is a multisampled pixel that contains information of color, z-value and texture data. In addition the PS can receive other values from earlier in the pipeline, like the normals. The PS-programs runs on all pixel fragments independently and in parallel using the unified shader processors on the GPU.

A PS can be used to manipulate lighting in each pixel, where an example is by manipulating the normals with a bump map or a parallax map. In that way the objects on the screen can look much more detailed than without, where effects like human skin can get pores, a chair can get the leather seat look, etc. can be made. A PS can also use the frame buffer to do different digital image processing functionality, explained further in Chapter 4. Typical digital image processing manipulations can be gaussian blur, explained in Section 4.2.

Figure 2.6: Pixel Shader blurring techniques. The screen is taken with permission from [8].

2.3 GPGPU

There are a few difficulties with the traditional way of doing GPGPU. With the graphics API overhead that are making unnecessary high learning curve and making it difficult to debugging. In NVIDIA CUDA (Compute Unified Device Architecture) and Microsoft DirectX Compute Shaders, programmers have direct access to the hardware for better control. A programmer also does not need to use the graphics API. These GPGPU specialized programming languages focuses on a low learning curve for developing applications that are scalable with the increase number of processor cores.

Further in section 2.3.1 we will give a summary of the hardware dependent NVIDIA CUDA programming model, which are picked up from Eirik Ola Aksnes' and Henrik Hesland's paper GPU Techniques for Porous Rock Visualization [1].

2.3.1 NVIDIA CUDA

The latest generations of NVIDIA GPUs are based on the NVIDIA Tesla architecture that supports the NVIDIA CUDA programming model. The NVIDIA Tesla architecture is built around a scalable array of Streaming Multiprocessors (SMs). Each SM consist of several Stream Processors (SPs), that have two Special Function Units (SFU) for trigonometry (sine, cosine and square root), a multithreaded instruction unit, and on-chip shared memory [10], as illustrated in Figure 2.7.

To a programmer, a system in the NVIDIA CUDA programming model consists of a host that is a traditional CPU and one or more computes devices that are massively data-parallel coprocessors. Each device is equipped with a large number of arithmetic execution units, has its own DRAM and runs many threads in parallel. The NVIDIA CUDA devices support the SPMD model where all threads execute the same program although they don't need to follow the same path of execution. In NVIDIA CUDA, programming is done with extension ANSI C, allowing the programmer to define data-parallel functions, called a kernel that runs in parallel on many threads [32]. Parts of programs that have little parallelism executes on the CPU, while parts that have rich parallelism executes on the GPU.

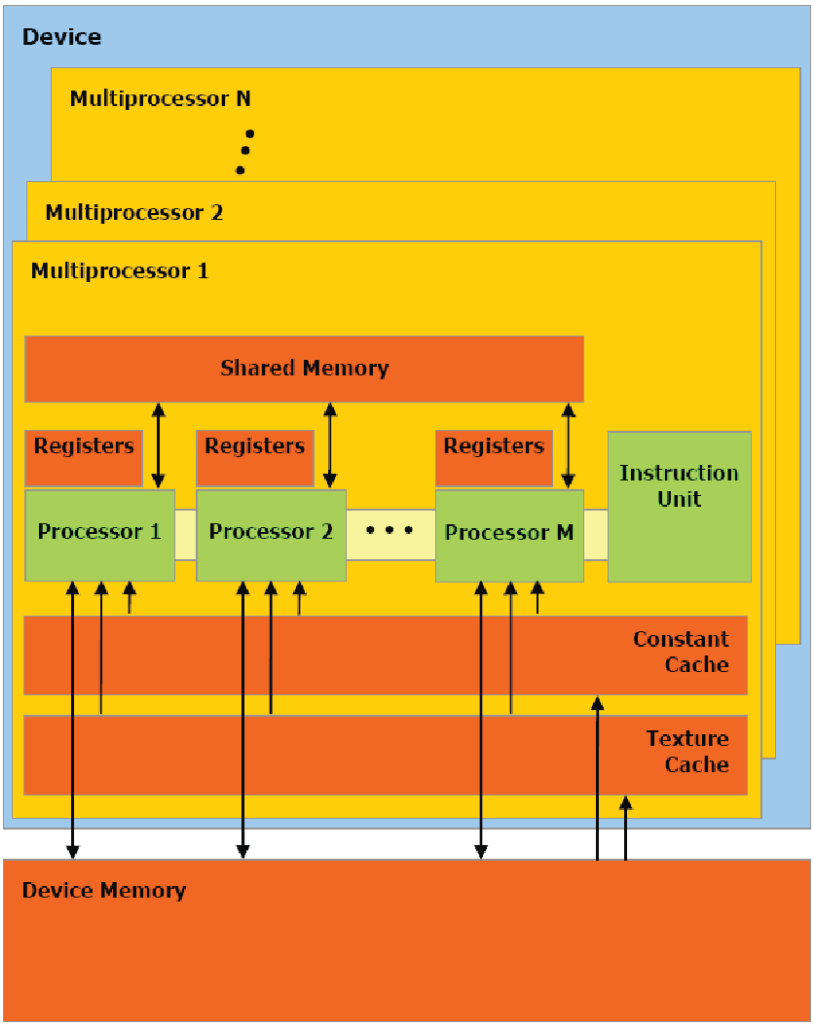


Figure 2.7: The NVIDIA Tesla architecture. Figure is taken with permission from [10].

GPU threads have very little creation overhead and it is possible to switch between threads that execute with near zero cost. The key to performance in NVIDIA CUDA is to utilize massive multithreading, a hardware technique which run thousands of threads simultaneously to utilize the large number of cores and to overlap computation with latency [39]. Under execution threads are grouped into a three level hierarchy, as illustrated in Figure 2.8. Every kernel executes as a grid of thread blocks, where each thread block is an array of threads that has a unique coordinate in the kernel grid. The individual threads have a unique coordinate in the thread block. Threads within the same thread block can perform synchronizing.

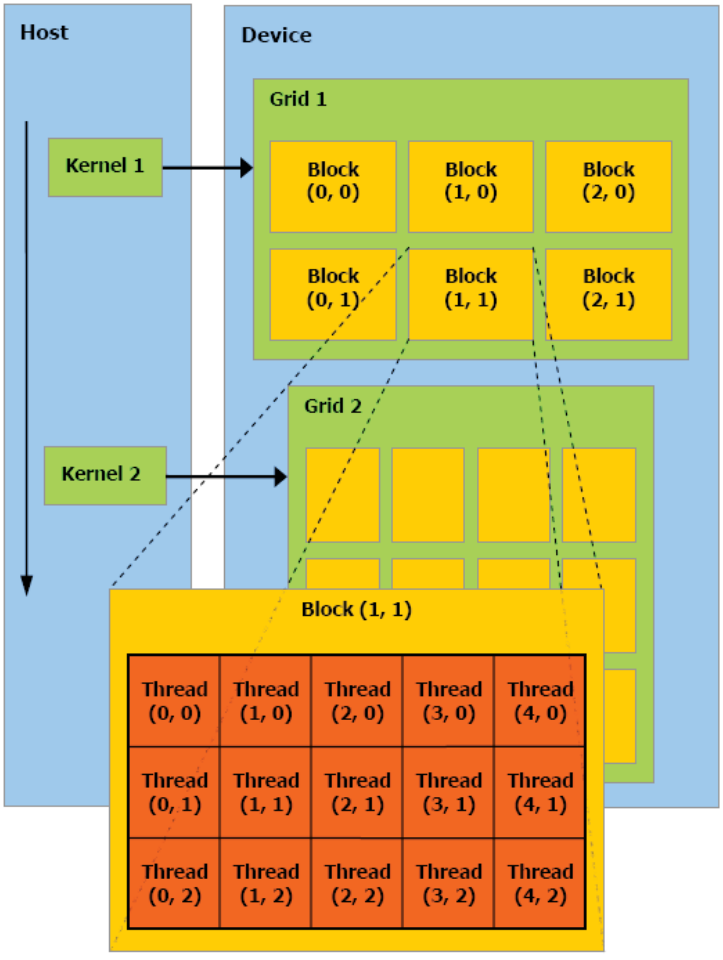


Figure 2.8: The NVIDIA CUDA thread hierarchy. Figure is taken with permission from [10].

All threads in NVIDIA CUDA can access data from diverse places during execution, as illustrated in Figure 2.9. Each thread has its private local memory and the architecture allows effective sharing of data between threads inside a thread block, by using the low latency shared memory. Finally, all threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads, the texture and constant memory spaces. Those memory spaces are optimized for various memory accesses patterns [10]. The CPU can transfer memory to and from the GPUs global memory using API calls.

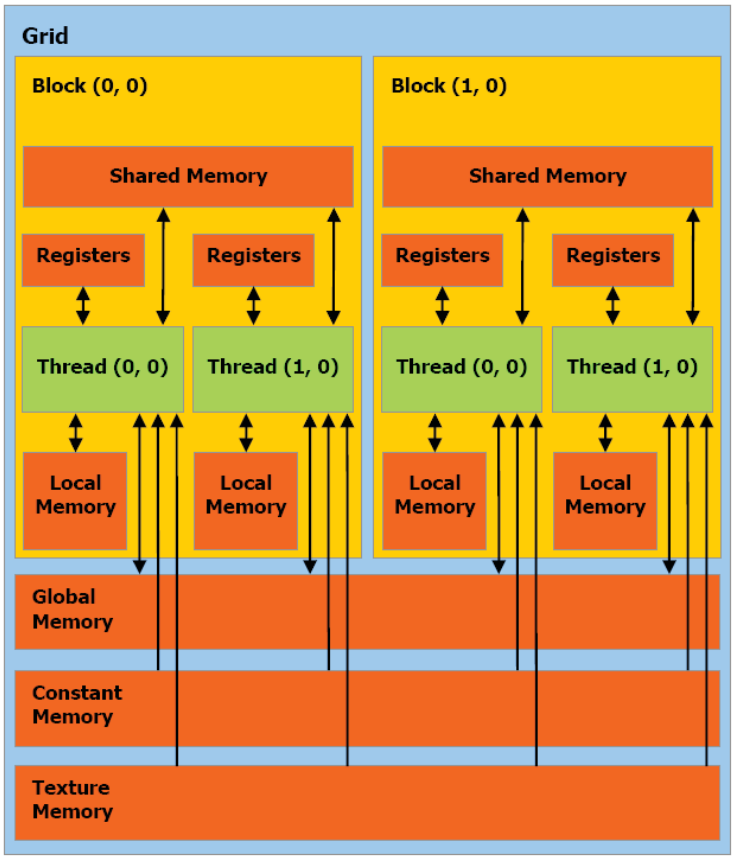


Figure 2.9: The NVIDIA CUDA memory hierarchy. Figure is taken with permission from [10].

Each SM can execute eight thread blocks at the same time. There is however a limit on how many thread blocks a SM can process at once, one need to find the right balance between how many registers per thread, how much shared memory per thread block and the number of simultaneously active threads that are required for a given kernel [32].

When a kernel is invoked, thread blocks from the kernel grid are distributed to SM with available execution capacity. As one block terminate, new blocks are lunched on the SM [10].

Under execution threads within a thread block grouped into warps. Warps are 32 threads from continuous sections of a thread block. Even though warps are not explicit declared in NVIDIA CUDA, knowledge of them may improve performance. The SM executes the same instruction for every thread in a warp, so only threads that follow the same execution path can be executed in parallel. If none of the threads in a warp have the same execution path, all of them must be executed sequential [32].

2.4 Graphics Programming

In an implementation of a 3D computer graphics program, there is needed a lot of functionality. Everything from calculating the colors of vertices, dependent on their normals, material type and light-positions, to calculations of how the representation will be shown on the screen in proportion to "eye"-coordinates (the location of camera in the scene) and what kinds of view (if the viewer has eyes like a fish, looking almost 360 degrees, or a camera, looking about 90 degrees). All this functionality is normally computed each frame, where a good frame rate is at least 30 frames per second (FPS). Most of these computations takes place in the graphics pipeline, and the programmer do not need to know how everything in the pipeline works, to implement a good solution. To use this pipeline efficiently, we need an Application Programming Interface (API), where the two most common is OpenGL and Microsoft DirectX, which will be explained further in Sections 2.4.2 and 2.4.3.

2.4.1 Buffer Objects

In graphics programming we often use buffer objects as a way to store data on the GPU global memory. There are three main types of these buffer objects;

Vertex Buffer Objects (VBO), Pixel Buffer Objects (PBO) and Frame Buffer Objects (FBO). These are basically arrays for storing data on the GPU to make it easy to use and easy communicate through.

When we are computing image processing techniques, we can use PBO or FBO for storing the image. In our case the image is stored by creating a PBO in OpenGL, then use NVIDIA CUDA functionality to map the PBO back and forth between OpenGL-code and NVIDIA CUDA-code. The PBO will then be made available for manipulation on each pixel when computing NVIDIA CUDA code on the GPU, and then later be available for rendering in the OpenGL code. The precise same method can be used for the other types of buffer objects, and is the main communication source between NVIDIA CUDA and OpenGL or Microsoft DirectX, without letting the information back and forth to the CPU.

2.4.2 OpenGL

OpenGL (Open Graphics Library) is a cross-platform and hardware independent graphics API, developed by Silicon Graphics Inc (SGI) in 1992, and can be used both for 2D and 3D graphics applications [20]. OpenGL contains a large set of commands, where some are doing things like drawing simple objects, like points of triangles and others are doing light calculations or simply opening a window. A typical OpenGL program begins with opening a window in the framebuffer. To use the full graphics pipeline, with all the shader-stages, OpenGL needs the add-on GLSL. GLSL 1.30 supports pixel and vertex shaders. And with another add-on, the geometry shaders are also supported, thou OpenGL are designed to be a 3D rendering system with the purpose of being accelerated by the hardware.

Some advantages with OpenGL are the communication with other APIs, like OpenCL (Open Compute Language) which is becoming very important for performance based applications. The OpenCL support was implemented in OpenGL 3.1 released 24. March 2009. OpenGL can also be programmed in most programming languages, and is a very flexible API.

2.4.3 Microsoft Direct3D

Microsoft Direct3D is a graphics API, which is a part of Microsoft's DirectX API. Direct3D is designed to be a 3D hardware interface, which is communicating directly with the hardware. The API is available only on

Microsoft Windows operating systems and the gaming consoles from Microsoft (Xbox and Xbox360). Included in Direct3D, is all functionality in the graphics pipeline and all SM5 was introduced in DirectX 11, using HLSL for programming the shader stages. For a computer running a Direct3D application with functionality the GPU do not support, these functions will be emulated with software rendering. An example of this if an application using the tessellation stage of the pipeline is run on an older GPU, the CPU will compute those results, and the framerate will sink dramatically.

DirectX 11 also includes functionality for rendering multithreaded, meaning threads on a multicore CPU can render to the same Direct3D device. Another functionality included in DirectX 11 is the compute shaders, which is much of the same as NVIDIA CUDA. NVIDIA CUDA is explained further in Section 2.3.1.

Chapter 3

Porous Rock Structures and Volume Rendering

The word petroleum, meaning rock oil, refers to the naturally occurring hydrocarbons that are found in porous rock formations beneath the surface of the earth. Petroleum is the result of millions of years of heat and pressure to microscopic plants and animals. Oil does not typically lie in huge pools, but rather within rocks or sandy mud. A petroleum reservoir or an oil and gas reservoir is typically an underground accumulation of oil and gas that are held and stored within porous rock formations [13].

The challenge is how to get the oil out. The recovery of oil involves pumping water (and sometimes other chemicals) to force the oil out and the bigger the pores of the rocks are the easier it is. Not just all rocks are capable of holding oil and gas. A reservoir rock is characterized by having enough porosity and permeability, meaning that it has sufficient storage capacity for oil and ability to transmit fluids. It is vital for the oil industry to analyze such petrophysical properties of reservoirs rocks, to gain improved understanding of oil production.

3.1 Core Analysis

Information gained through core analysis is probably the most important basic technique available for petroleum physicists to understand more of the conditions that affect production [21]. Through core analysis the fluid characteristic of the reservoir can be determined. While traditional core analysis returns valuable data, it is more or less restricted to 2D description in form of slices, and does not directly and accurately describe the 3D nature of rocks

properties [22]. With the use of microcomputed tomography geoscientist can produce high-resolution 3D representations of the internal pore structures of reservoir rocks. It is a nondestructive method to determine petrophysical properties such as porosity and permeability [21]. Properties used further as inputs into reservoir models, for numerical simulations to estimate the recovery factor.

3.2 Computed Tomography

Computed Tomography (CT) is a non-destructive imaging-method particularly used in medical imaging, where it is used to examine the internal structures within a human body. The first prototype of the CT-scanner was made by Hounsfield in 1967, and over the years the technology has advanced rapidly, where resolution and speed has been the most important improvements. In addition to the CT-scanners importance in the medical field, geoscientists started using them in the mid 1980s, which gave the geology-field a nondestructive evaluation method of core samples [21, 1]. The technology used today in the geologic field is called Microcomputed Tomography (μCT), which is a small CT-scanner with higher resolution than the normal CT-scanners. The high resolution of μCT makes a highly detailed volume of the samples, and will make it possible to observe even smaller pores than before.

A CT-scanner is a special type of X-ray machine, with the same components: an X-ray source/tube, the object to be scanned and a detector array, as in Figure 3.1 [9]. In addition, the CT-scanner has a way to rotate the sample (in scanners for core-samples) or the X-ray source and the detector (in medical imaging). The X-rays are being emitted in a fan-shaped beam, where the rays are being absorbed differential through the material [14]. Since rays are absorbed differently in proportion to the material densities, there will be possible to observe the differences between pores and rock in the resulting image.

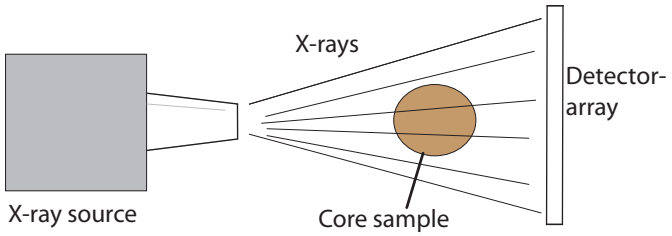


Figure 3.1: X-ray attenuation measurement process.

Since either the object to be scanned or the X-ray source and the detector can rotate, a CT-scan will generate several X-ray images 360 degrees around the object, as in Figure 3.2, which can be processed to make a digital 3D representation of the scanned object [21, 1, 9]. In the volume procession, there will first be generated image slices of the volume (as illustrated in Figure 3.2), which resulting can generate a 3D-model of the sample.

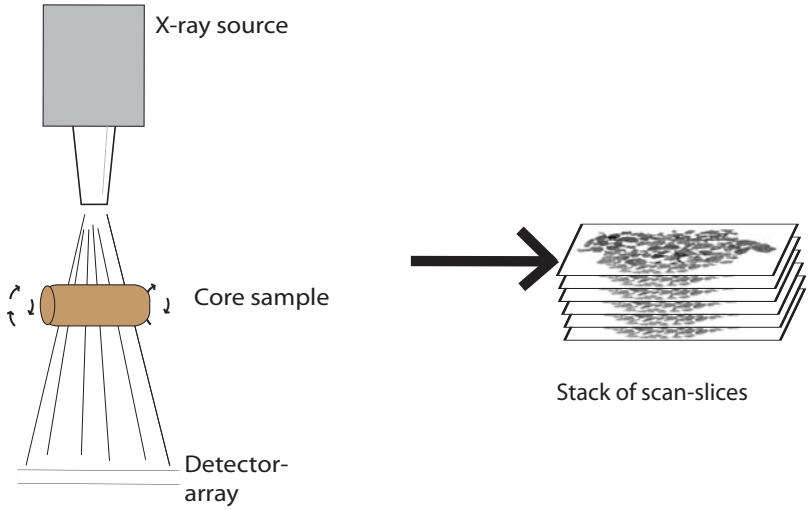


Figure 3.2: CT scan process with rotating core sample.

In this thesis, all scan data is created from a SkyScan 1137 μCT scanner in the Department of Petroleum Technology, NTNU. The scanner uses a computer cluster to reconstruct the scan dataset in less time than the scan duration. And the resolution of a dataset created can be up to 8000 x 8000 pixels per image, extremely detailed, but taking a huge amount of storage space.

A problem with a CT-scan is that it produces various types of artifacts, which will be explained further in Section 3.2.1.

3.2.1 Typical Artifacts in CT-scans

An artifact in CT is the difference between the reconstructed image and the true attenuation coefficient of the object. Artifacts can therefore significantly reduce the quality of CT-images, sometimes as much that the image gets unusable for medical purposes or fluid simulation, and can often be mistaken for a disease in the medical field. There are mainly two classes of artifacts in CT-images, which can occur in the scanning of core samples. The classes are physics-based and scanner-based artifacts. In addition to these two classes, medical imaging meets a big challenge in patient-based artifacts, which can occur by factors as patient movement, and helical and multisection technique artifacts, which come from the reconstruction process of the slices. Some of the artifacts are minimized by modern CT-scanners and their scanner software. To reduce these artifacts and optimize the image quality, there is important to understand why and how these artifacts occur [4]. Further is an overview of the most important artifacts for scanning of core-samples.

Physics-based artifacts

Physics-based artifacts are a result from the physical process involved in the acquisition of CT data [4]. In an X-ray beam, there are many individual photons, where each have their own energy. The energy level can vary a lot from photon to photon. When the beam is omitted through an object, the lower energized photons will be absorbed as the material gets harder (the density gets higher); this technique is called beam hardening.

When an X-ray passes through the middle of a circular shaped object, it passes through more material, than the other rays. Therefore in the middle, the beam becomes harder. More attenuation occurs in the center of the object, than around the edge, and the resulting density will increase in the middle, even if there is the same material around the edges, as illustrated in Figure 3.3. This artifact is called a cupping artifact [4].

A way to reduce the beam hardening artifacts is to place a filter, usually a flat piece of some metallic material, which the beam has to pass through before the object to be scanned. This will filter out the lower energy photons, and is called "pre-hardening". In addition, CT-scanner manufacturers often

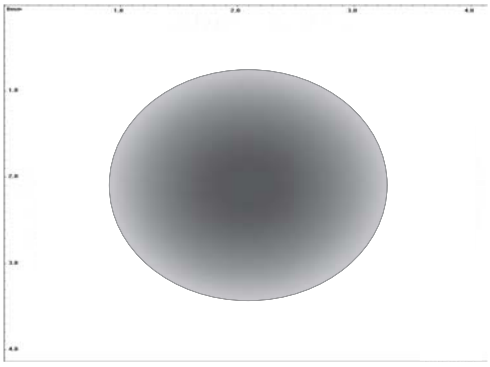


Figure 3.3: Cupping artifact.

deliver a beam hardening correction in their software [4].

Another type of physics-based artifact is the partial volume artifact. These occurs when an off center dense part of the scan is partially scanned by the beam, like in Figure 3.4 [4]. Since the X-ray beam is fan shaped, the beam can miss a hard part of the rock when scanning on one side, but find the same part when the beam is sent 180 degrees of the missing position. By doing this, the artifacts will appear as a shading in the image, like Figure 3.4. This effect can easily be avoided by putting the object to be scanned far enough away from the X-ray source, which the object will be fully inside the beam, the whole rotation process.

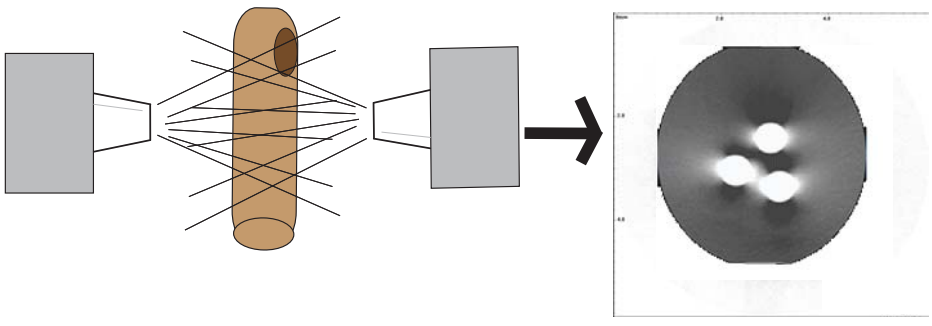


Figure 3.4: Partial volume artifact.

Scanner-based artifacts

CT-scanners are very sensitive to the position of the X-ray tube and the detector array. Due to defective detector elements or shifts in the output from individual detector elements ring artifacts can occur, which is both complete and incomplete circles, in part of or all over the scandata image. Illustrations of ring artifacts can be seen in [4]. Ring artifacts can also occur from imperfections or variations in the incoming beam, or from variations in the beam together with the point-spread function of the detector elements.

A common method to reduce the ring artifacts is flat-field correction, which is a method who uses an image without the sample and an image without the beam, to generate the real image [3]. Even thou the rings are reduced, they will not completely disappear from flat-field correction. Another method to reduce the rings, is by filtering methods before the image reconstruction, which is further explained in [30], or after the reconstruction [3, 35]. In practice, no scanner is completely free of geometric misalignments, where the X-ray source, the turn table for the object to be scanned and the detectors all have some misalignments compared to each other. A technique to reduce the effect by the misalignment is by calculating the perfect positions of these elements, which is explained in [38].

Most Micro-CT systems, like the SkyScan 1076 system, are highly sensitive to slight difference in the sensitivity of adjacent detector elements. For scans that need high contrasts, the resolution is turned up, and the ring artifacts will increase [35].

3.3 Volume Rendering

As explained in [1], volume representation techniques are being used in a variety of areas, ranging from medical modeling of organs and bone structure to seismic and geological representations of pore-geometry, and many other areas where representation of Magnetic Resonance Imaging (MRI)/CT scan data or deformation of 3D objects is needed. Traditionally these techniques used sequential calculations on a CPU, or parallel computations on CPU clusters, before rendering on the GPU. The generation of volume models with a high level of complexity is a highly parallel task, where lots of independent vertices can be generated simultaneously.

Scan data from CT or MRI is often represented as a stack of 2D images,

where each image represents one slice of the volume. The volume can be computed by extracting the equal values from the scan data, and rendering them as polygonal isosurfaces or directly as a data block.

A volume can be completely described by a density function, where every point in the 3d-space is represented by a single scalar-value. Typically the values higher than a threshold-value is hard rock and the points having a value under the threshold are fluids like oil, gas or other materials with a lower density than the material we want to find. In the 3D model, there is placed a boundary straightly at the threshold, to separate rock from fluid.

The most common algorithms for computing the isosurface is The Marching Cubes (MC) and Marching Tetrahedron (MT). These two algorithms are almost the same, where both of them are using voxels, where each corner gets a binary, if it is inside or outside the threshold. By using these binaries as a sequence, we can look up in a precalculated table, which tells us how to generate polygons in that voxel. The difference between the two is how a voxel is represented, where MC is using 8 points at a time, forming a cube, and MT is using 4 points forming a tetrahedron. A more detailed explanation of these is explained in [1].

There also exist some other techniques for computing a volume representation. To mention some of them, volume ray casting, splatting, shear warp, texture-mapping and hardware accelerated volume rendering. They all require color and opacity for each sample value. If you want an overview of what each of them are, take a look at [43, 33].

Chapter 4

Digital Image Processing

Today's technology makes it possible to manipulate images with digital techniques. A digital image can be viewed as a two-dimensional function f(x, y), where x and y are coordinates in the spatial plane (image plane) for a given pixel and f represents the intensity of the pixels color values. A pixel's given color can be represented by one digit if the image is a grayscale image and 3 or 4 digits if the image has full colors. A typical 4 digit image representation is 32 bit, using 8 bits pr color channel, which is red, green, blue and alpha (RGBA). Each of the color channels in 32 bit RGBA-images use a value between 0 and 255 for each color. The first three channels (RGB) represent the color of the pixel, where the three values corresponds to a position in the color space, like illustrated in Figure 4.1 (the white corner is 255 in all 3 color values and the black corner has the same values at 0). The alpha channel represents the opacity of the pixel (which can be handy for a texture in a 3D scene for showing transparent materials).

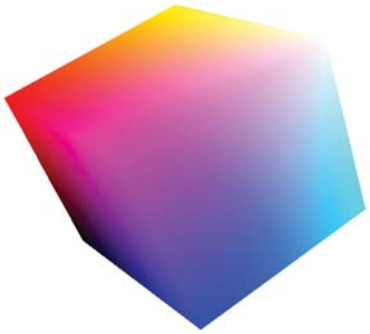


Figure 4.1: The RGB 3D space.

The area of digital image manipulation is called Digital Image Processing (DIP), and is divided in three main categories [18]. The first category is called image processing, where some manipulations are done on an image and sends the result image as an output, like Figure 4.2b. The second category is image analysis, which extracts information out of the image, an example of information is segments or edges, like illustrated in Figure 4.2c. The last category is image understanding/computer vision, where the input image is analyzed and returns a high level of description. In an image analysis, there is important to distinguish between the objects of interest and the rest, the techniques for finding the objects are often referred to as segmentation, where two of the most important techniques is thresholding (Section 4.3 for more details) and edge detection (Section 4.4 for more details). An important thing to understand when working with image segmentation is that there are no universal techniques that will work on all images and no techniques is perfect.

Figure 4.2: Digital image processing categories.

DIP can not only work on images able to be seen by human eyes, but also images from almost the entire electromagnetic spectrum (illustrated in Figure 4.3), ranging from gamma to radio waves [18]. Therefore DIP is important in many areas, from medical or geological CT-scans to noise reduction in radio waves.

By doing DIP, there is several different start approaches, the manipulation can be done in the spatial domain, which is working with the pixels as they are. Another approach by is using the images frequency domain

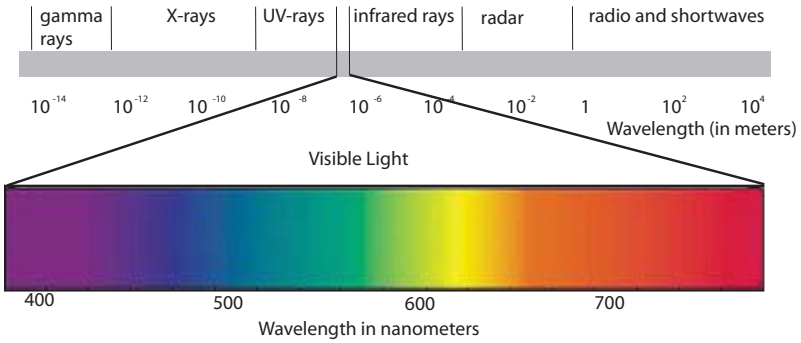


Figure 4.3: The electromagnetic spectrum.

(Fourier domain), where high frequencies (sharp edges) from the original image is concentrated in the middle, and lower frequencies (colored surfaces) are represented longer out from the middle of the frequency image, which are illustrated in Figure 4.4. By using the frequency domain, lots of manipulations is opened in addition to the ones in the spatial domain, an example of such a manipulation is filtering methods for filtering out lower frequencies, which will result in an edge sharpening method. You can read more about the frequency domain in [18].

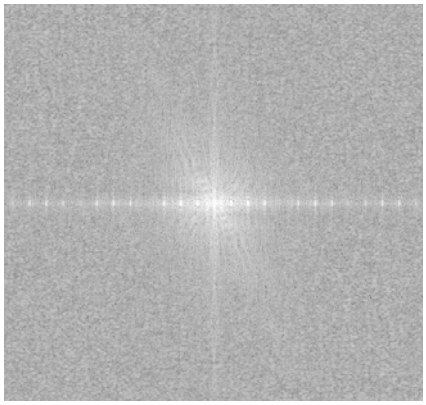


Figure 4.4: Frequency image.

4.1 Logical Image Operators

All DIP algorithms need to involve logical functions operating on vectors of logical variables [40]. Logical Image Operators works just like other mathematical functions, where an example is that sometimes we need to divide two images together to create one new image. The images then have to be the same size, or where a corner of the smallest image starts in the other image needs to be specified. Then the pixels with the same reference in both pictures can be used in a logical operation. Dividing the two of them and creating the image c = a - b, where an example of this is shown in Figure 4.5. In the same way we can use most other logical operators too.

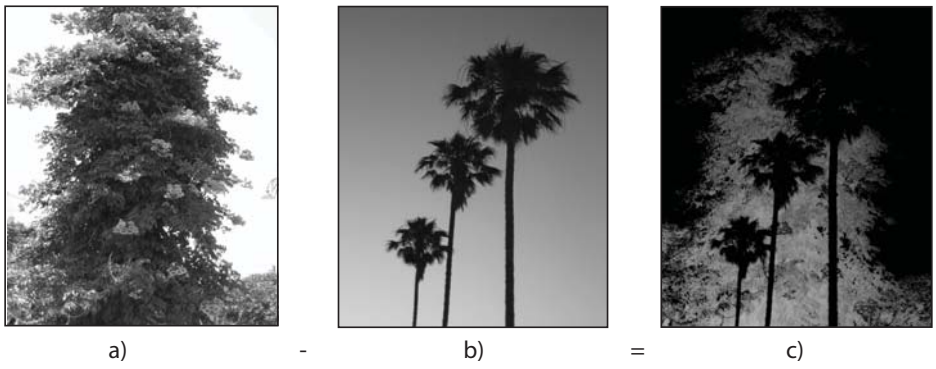


Figure 4.5: Mathematic operator on images.

4.2 Spatial Filters

A spatial filter is a filter/mask doing operations on an image, by working in neighborhoods of pixels. These techniques are highly parallelizable, where they operate with an input image and a mask, and every computation outputs a pixel value for the output image. A spatial filter can be defined by:

$$g(x, y) = T[f(x, y)] \tag{4.1}$$

Where g(x, y) is the resulting image, f(x, y) is the input image and T is the operator operating on f and the neighborhood defined around the point

(x, y) [18]. A mask defines both the size of the neighborhood and the operations to do, and is in most cases much smaller than the image. The operation is done in the whole neighborhood of the input image, and results one pixel in the output image.

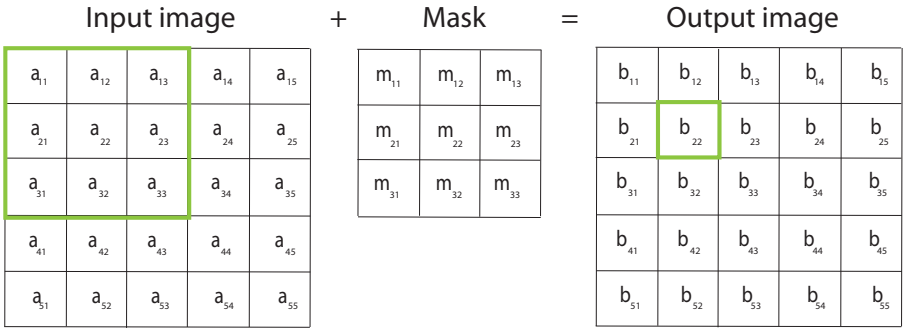


Figure 4.6: Mask operations by using Equation 4.2.

$$
\begin{aligned}
b_{22} = & \\
& (a_{11} * m_{11}) + (a_{12} * m_{12}) + (a_{13} * m_{13}) + \\
& (a_{21} * m_{21}) + (a_{22} * m_{22}) + (a_{23} * m_{23}) + \\
& (a_{31} * m_{31}) + (a_{32} * m_{32}) + (a_{33} * m_{33})
\end{aligned}
\tag{4.2}
$$

An example of a mask is a Gaussian blur mask. Where the mask illustrated in Figure 4.7 is moved over and computed for each pixel-location it fits in the input image, to create the output image, as illustrated in Figure 4.7, where we can observe a smoothing compared to the original image. Another example of a filter is the Sobel edge detector, which finds edge information in the image. Sobel is explained further in Section 4.4.1.

4.3 Thresholding

Thresholding is a simple and computationally inexpensive segmentation method. Thresholding uses a threshold-value (border-value) to decide the inside and outside of an object in the image, or in some cases differentiate between foreground and background. The threshold value is then compared to each

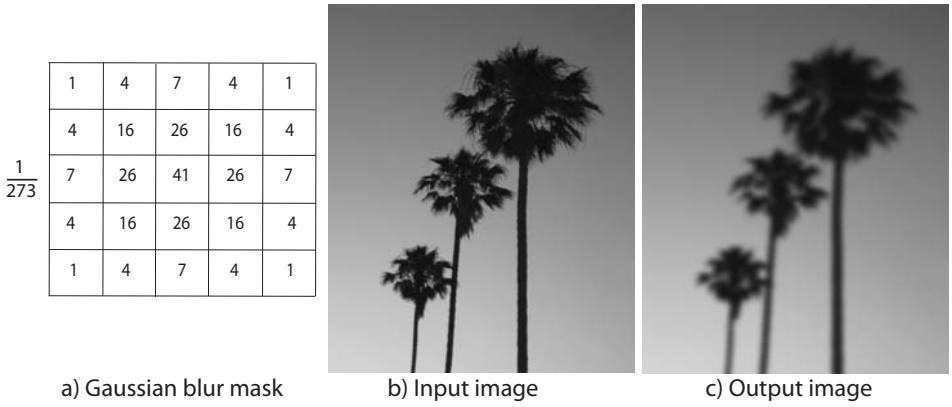


Figure 4.7: Gaussian blur mask operation.

pixels intensity value, where a simple global threshold works this way:

```
foreach (pixel f(i,j) in image f)
        {
            if (f(i,j) >= threshold_value)
            then g(i,j) = f(i,j)//where g is the object represented, and
                                //the pixels not in g is the background pixels.
        }
```

To get a good segmented image from using a threshold, the threshold value is really important. There are mainly two different methods of choosing the threshold value: manually chosen by the user or automatic algorithm for computing the value (called automatic thresholding). In some images a threshold value can be chosen by making an intensity histogram, the images which then are easy to segment will return a histogram with the objects of interest in one end of the intensity scale and the background at the other end, with a valley between, illustrated in Figure 4.9.

There exist some different types of thresholding, where a variable threshold is a threshold which changes through the image. Another type is adaptive threshold, which uses local thresholds for different regions in the image.

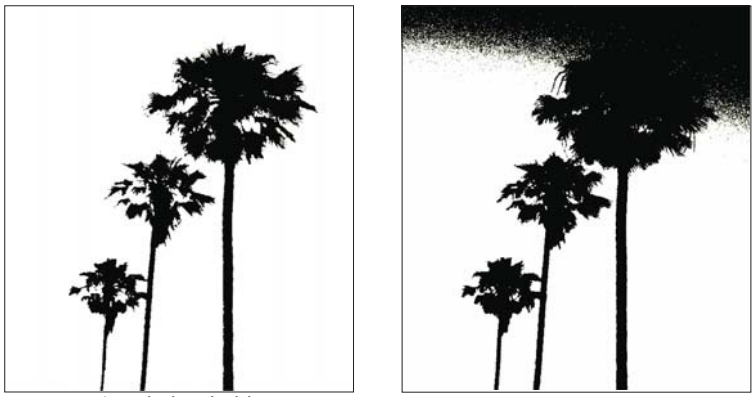


Figure 4.8: High and low thresholding.

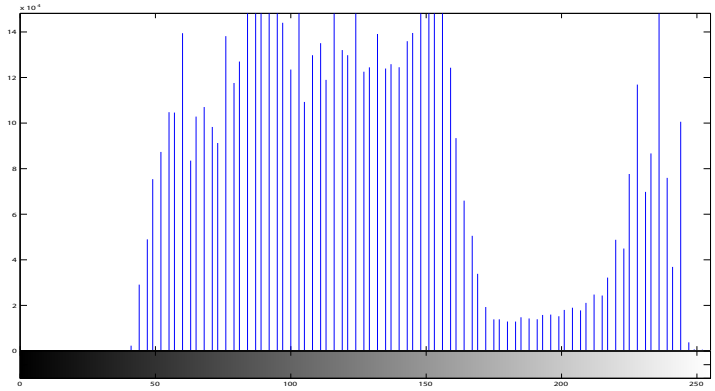


Figure 4.9: Histogram of Figure 4.7b.

4.4 Edge Detection

Important objects or events in an image are often represented by a sharp change in intensity values. In Figure 4.10a we can observe a large intensity change between the sky and the palm trees, we also can observe intensity changes in the sky itself. In image processing, edge detection is a class of algorithms which can find the edges between these intensity changes, and ideally lead to connected boundaries of objects and most of the less important information filtered off. Real life images, like Figure 4.10 is in most cases too complex to get a perfect result from these edge detection algorithms. Real life images can also contain artifacts, which make it even harder to get perfect edge detection. The artifacts of the images can be ring artifacts or image hardening by using a CT scanner (Section 3.2.1 for more information)

or reflection and shadows and many other types of challenges, which make DIP even harder [18].

Figure 4.10: Edge detection.

To capture the intensity changes in an image, the edge detection algorithms mainly uses two different methods; first order derivatives of the image, finding maximum and minimum points in the derivate image. The other approach is by finding the zero-crossings in the second order derivatives. By derivation of an image, the edge detection algorithms use one or several filters.

First order derivative methods detect the edges by measuring the strength of the gradient, which is called edge strength. This is normally done by using two filters, finding the derivatives of the gradients in two directions 90 degrees on each other. First order derivative can be given by:

$$\frac{df}{dx} = f(x+1) - f(x) \tag{4.3}$$

Where f(x) is the intensity of a pixel number x in the given direction. One of the most used first order derivative methods is the Sobel filer, which is explained further in Section 4.4.1.

The zero-crossing-based methods normally need one filter to find the second order derivative in all directions. The second order derivative is given by:

$$\frac{d^2f}{d^2x} = f(x+1) + f(x-1) - 2f(x) \tag{4.4}$$

Where at least 3 pixel-values is needed to find an edge. One of the basic zero-crossing based methods, which many other methods is based on is the Laplacian filter, illustrated in Figure 4.11. These second order derivative methods can also mark the zero-crossings differently, where edges from high to low intensity and the edges from low to high intensity can be differentiated.

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	24	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

Figure 4.11: The laplacian filter mask.

The main difference between first and second order derivatives is that second order derivatives will find more details in the image, and therefore also will be more sensible to noise. A way to deal with the noise sensitivity is by using a Gaussian blur first, or combined called Laplacian of Gaussian (LoG). A problem with this approach is that it will have problems finding curved edges [18]. Another difference between the two types of approaches is when dealing with ramp edges, like the one illustrated in Figure 4.12 [18]. The first order derivatives will in this case draw a broad edge as seen in the image. With the second order derivative we will be able to find the centre between the two crossing points, since it can differentiate between the ones going from high to low intensity and the ones going from low to high intensity.

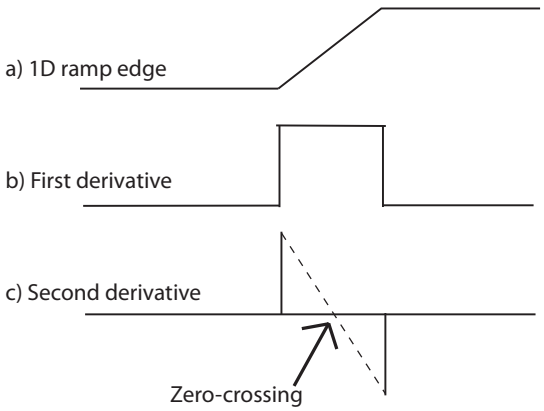


Figure 4.12: Ramp edge in 1. and 2. order derivative.

4.4.1 Sobel filter

The Sobel filter is the most used edge detection technique [18], which is based on first order derivate using two Sobel masks. The masks (given in Figure 4.13) is computed one by one in each direction, creating two gradient images, one for x-direction and one for y-direction. The masks can also be drawn diagonally, which will improve the diagonal edges a little, but make the straight edges slightly more inexact. As we can observe in the masks in Figure 4.13, each of them has a total weight of zero, which is required for a derivative operator. Another thing to observe is that the weight across the center of the derivative direction is two. The higher weight across the center of the mask provides an image smoothing, which will make the filter slightly more resistant versus noise [18].

-1	0	1
-2	0	2
-1	0	1

a) x-direction

-1	-2	-1
0	0	0
1	2	1

b) y-direction

Figure 4.13: The Sobel filter masks.

To compute the partial derivatives, we use normal mask calculation given,

by using the Sobel masks separately over an image a, as done in Equation 4.2:

$$g_x = \frac{df}{dx} = (a_{13} + 2a_{23} + a_{33}) - (a_{11} + 2a_{21} + a_{31}) \tag{4.5}$$

$$g_y = \frac{df}{dy} = (a_{31} + 2a_{32} + a_{33}) - (a_{11} + 2a_{12} + a_{13}) \tag{4.6}$$

Where g_x and g_y is the partial derivatives and a_{xx} represents pixels in the image, using the intensity values. After calculating the partial derivatives, the edge strength can be computed. This is normally done with one of the two methods given:

$$g = \sqrt{g_x^2 + g_y^2} \tag{4.7}$$

$$g \approx |g_x| + |g_y| \tag{4.8}$$

Equation 4.7 gives the most precise edge strength, but square roots are computationally expensive, and therefore in Equation 4.8 is often used in performance based applications. By using Sobel on an image, it will look like Figure 4.10b, with contour around the objects found.

Masks like Sobel, which are symmetric about a center point is excellent for computing edge directions. After the mask operations, we got two pictures of the original size, one with gradients in the x-direction and one with gradients in the y-direction. By using Equation 4.9 we can decide the direction of the edge. Normally the angle is being round off to the nearest 45 degrees, since there is only 8 pixels around a pixel, connected in 8-adjacent, which means the pixels can be connected corner to corner or side to side. More about the round-off of the angle will be explained in section 4.4.2.

$$\Theta = \arctan \frac{g_y}{g_x} \tag{4.9}$$

Where Θ is the orientation angle of the edge, and g_x and g_y is the partial derivatives in the given directions.

4.4.2 Canny edge

Using the Sobel edge detector by itself on real world images will give a noisy resulting image. This is because of three main problems [18, 36]:

- Some important edges will get low edge strength; this will cause missing edges when using a threshold on the edge strength image. The balance by using a lower threshold will cause edges which are not true edges to appear as noise.
- Often there will be found multiple responses to a true edge, this will cause thick edges and in detailed images will make the result inexact.
- Noisy single pixels can be found at the edge border, which will cause missing pixels in the resulting edges.

The Canny edge detector, developed by John F. Canny in 1986 [6], is known as a superior edge detector [18]. It is based on a five-step algorithm, which takes the three problems described over into consideration. Each step will now be explained:

Step 1: Gaussian blur

First we want to remove some noise from the image. By using a Gaussian blur mask, as explained in Section 4.2, the third problem described above will be reduced, and the noisy pixels will have lesser effect on the resulting image.

Step 2: Sobel edge detection

By using the Sobel edge detector, explained in Section 4.4.1, will give the gradients in x- and y-direction in addition to the edge strengths. Totally the first two steps will be a first order derivative of a Gaussian.

Step 3: Edge direction

To find the edge directions, Equations 4.5 and 4.6, then 4.9 will be used. The edge directions will get Θ-values between 0 and 180 degrees. As seen on in Figure 4.14a, a pixel 'a' has eight neighbors, the edge direction then can be in four directions only, because it can only be a straight line in a pixel. Therefore the use of adjusted directional angles is needed, where edge directions between 0 and 22.5 and between 157.5 and 180 degrees, will be set to 0 degrees, and so on. Each of the four new directions is adjusted by the areas shown in the harmonic circle in Figure 4.14b [19]. The new directions are then 0, 45, 90 and 135 degrees.

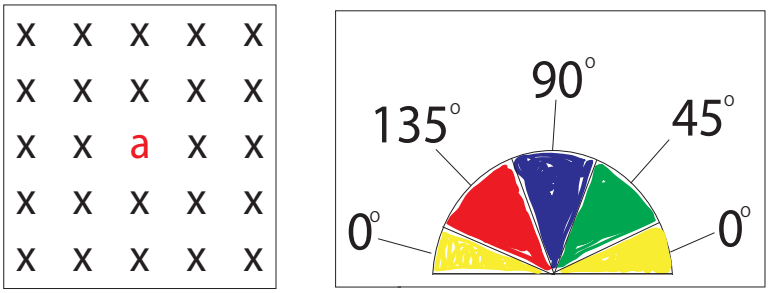


Figure 4.14: Edge direction adjustment.

Step 4: Non-maximum suppression

For the second problem described above, where more than one pixel represents one true edge, the use of non-maximum suppression is important. Non-maximum suppression is first considering each pixel in the image, comparing them to the two pixels 90 degrees on the pixels edge direction. If one of the two neighbor compared with has a larger edge strength, the current pixels edge strength is set to zero. This will lead to thin edges, or at least a reduced thickness of the edges.

Step 5: Hysteresis thresholding

To get a usable image after edge detection, thresholding is needed. For the first problem described above, where a global threshold value would cause either real edges to disappear or noisy pixels to appear, hysteresis thresholding is a good solution. The concept is using a higher and a lower threshold

value, where each pixel is compared to the higher threshold. For each pixel which has an edge strength higher than the higher threshold is presumed to be an edge pixel. Then, for each pixel connected to the edge pixels, using the edge directions and following the edge, which have edge strength greater than the lower threshold, is also marked as an edge pixel.

After using the five steps of the Canny edge detector, the three problems of using only the Sobel edge detector, as described above, is reduced significantly. In Figure 4.15 we can observe the difference between the Sobel Edge with global threshold and Canny Edge.

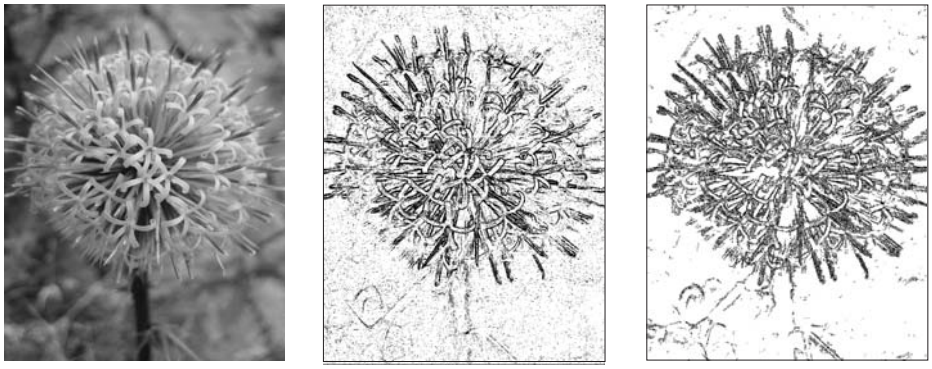


Figure 4.15: Sobel edge detection VS. Canny edge detection.

Chapter 5

Implementation and Guidelines

In this chapter, we will explain some thoughts around the implementation, the reasons of some of the algorithmic choices and some guidelines to follow when programming NVIDIA CUDA.

5.1 Platform and Hardware Specification

Our application is targeted towards graphics enabled desktop computers and workstations. Since it is a highly parallelizable application requiring a high framerate, the computations is needed to be calculated on a GPU. The GPU computations is implemented with NVIDIA CUDA 2.1, because of a low learning curve and further development of the volume rendering project done by Eirik Ola Aksnes and Henrik Hesland in the project GPU Techniques for Porous Rock Visualization [1]. At the moment NVIDIA CUDA is supported only by NVIDIA GPUs with SM4 support. Therefore the implementation also needs a NVIDIA GeForce 8800 GPU or newer.

For the graphical visualization the usage of OpenGL 3.1 is chosen, where the programming language is C/C++. C/C++ is chosen because of the importance of high framerate and easy communication with both OpenGL and NVIDIA CUDA. The compiler used for the implementation is Microsoft Visual Studio 2005.

The application is meant to prepare 3D models for reservoir applications such as Schlumberger Petrel. Schlumberger Petrel is a Microsoft Windows software application; therefore our application's file-functionality is specific for Microsoft Windows 32-bit systems, implemented by using the Win32 library.

5.2 Optimization Guidelines

As said in [1], to program efficiently on a GPU using NVIDIA CUDA, the knowledge of the hardware-architecture is important, that is explained further in the Section 2.3.1.

The primary performance element of the GPU, which really should be exploited by a NVIDIA CUDA programmer, is the large number of computation cores. To do this, there should be implemented a massive multithreaded program.

One of the problems with a GPU implementation is the communication latency. For computations, there should be as little communication as possible between the CPU and GPU, and often some time can be spared by doing the same computation several times, than load the answers between the CPU and GPU. The massive parallelization of threads is also important for hiding the latency.

A modern GPU contains several types of memory, where the latency of these is different. To reduce the used bandwidth, it is recommended to use the shared memory where it is possible. The shared device memory is divided into banks, where access to the same bank only can be done sequentially, but access to different banks can be done in parallel. Therefore the threads should be grouped to avoid this memory conflict.

Another function that should be used as little as possible is synchronization. This can cause many threads to wait a long time for another thread to finish up, and can slow the computation significantly.

The number of thread-blocks used simultaneously on a Streaming Multiprocessor (SM) is restricted by the number of registers, shared memory, maximum number of active threads per SM and number of thread-blocks pr SM at a time. Therefore the number of threads, memory used and total bandwidth should be configured carefully in proportion to each other.

5.3 Implementation

This section will describe the parts of our implementation. The programs functionality is meant firstly to load a selected dataset of bmp-grayscale im-

ages. After the loading, the user should be able to choose how many images to work with, up to maximum 20 images. The first image of the ones calculated for the user to work with will then show to the left of the screen. If one image were chosen, the middle image of the dataset will be used. The user should then be able by using the mouse pointer on the image, to localize the rock and the pores. We chose to let the user to be able to click on a predicted edge between the two different identifiers. It will not only be able for clicking, but also moving the mouse while holding the left mouse button, calculating the edges per frame. While moving, the resulting edge-image will not directly be saved, but the user will be able to move around until he or she is satisfied, and the new edges will be saved by unclicking the mouse button. When the user then is satisfied with the feedback of the edges, the user can click the next button, which replaces the current image with the next image to work with. Once the user is satisfied with all the images, the next thing to do is to create a 3D model with the thresholds chosen. The 3D model will then be shown on the right of the window, and the total graphical design will be as seen in Figure 5.1. When the 3D model is created, the user should also be able to export the 3D model as a file, and the file-type is importable in Schlumberger Petrel.

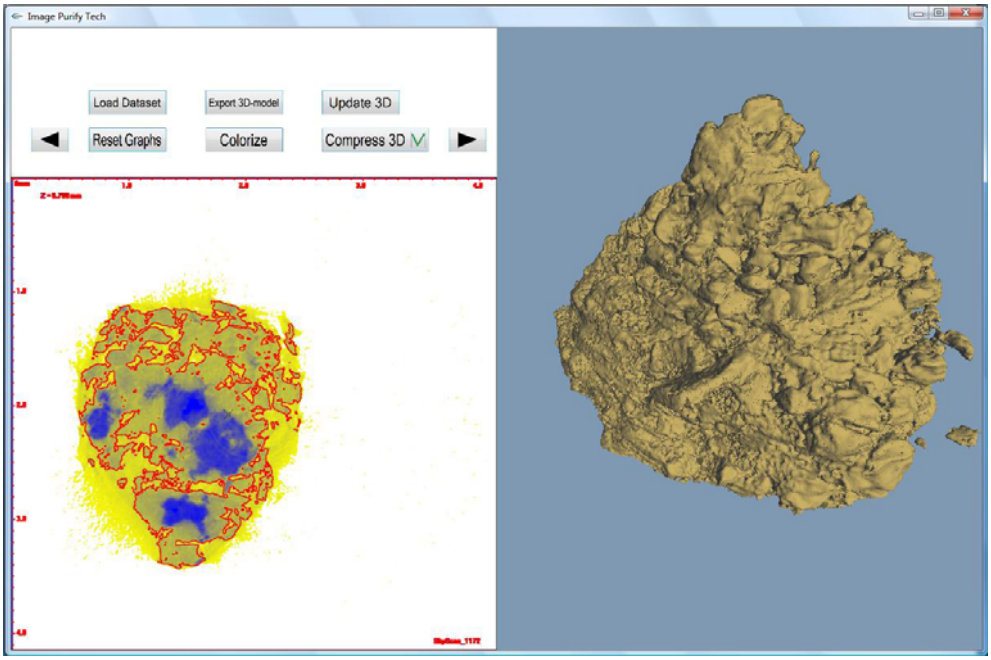


Figure 5.1: Our Graphical User Interface (GUI).

The threshold in the image has to be variable and different over different parts of the image. This is because the density values for pores in the center of the image are in some cases higher than the rock-densities in the outer parts of the image, as seen in Figure 5.2. From the image we can observe that there are 3 harder centers in the rock, with higher pore-densities around, than on the softer places of the scan. The pore shown has a maximum density of 72 and the rock shown has a minimum density of 64. The densities can also be different on different sides of the image; therefore we will have a look at a variable regional thresholding.

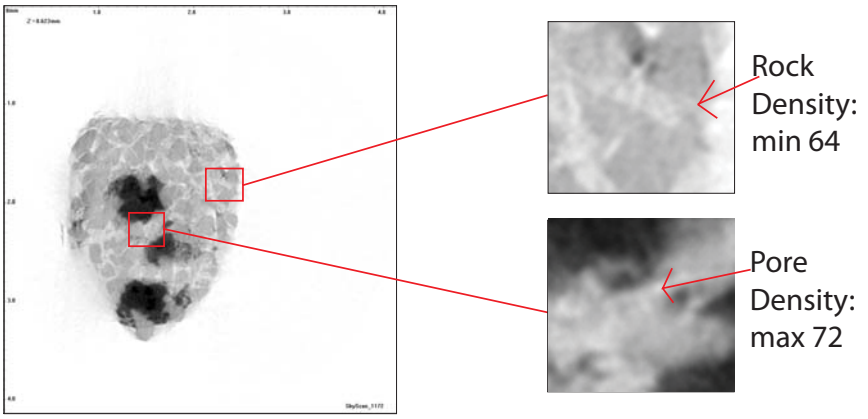


Figure 5.2: Comparing central pore and outer rock densities.

All the parallel algorithms for image processing and volume rendering is implemented in NVIDIA CUDA, and is executed on the GPU. The image processing parts could have been implemented in the pixel shader stage of the pipeline, handling a frame buffer object [26]. In addition our volume rendering algorithm Marching Cubes, could both have been implemented in the vertex shader [17] or the geometry shader stage [12]. The reason of why we chose to use NVIDIA CUDA on all these algorithms is the easier programming by using a GPGPU programming language.

5.3.1 The Graphical Display

The application needs two types of graphical rendering. 2D orthogonal rendering for the image display on the left and 3D rendering for the 3D volume rendering on the right. For the calculations of all graphics pipeline functionality, we used OpenGL.

For the 2D display we implemented texture support, since the images to work with is to be shown on the screen as a texture. By using the image as a texture, there also is implemented a way to zoom in on the images, for more accurate thresholding. To turn on 2D rendering, there is used an orthogonal viewport. In the 2D display we are also rendering the buttons, which are just 2D textures. For the events of the buttons, there is an invisible bounding box over the button texture, which reacts on mouse clicks.

The 3D display uses light calculations and depth tests, therefore it is important that either each polygon or each vertex has a normal vector. For rotating and moving the 3D volume around on the display, we are using the OpenGL matrix functionalities for rotation and translation.

5.3.2 Image Operations

Because of the scan data images uses 256 different grayscale values to represent the densities, some grayscale values can be difficult to differentiate for human eyes. This will be improved by a functionality explained in the Colorize image Section underneath. In addition, if the user prefers the grayscale image, or wonders how it looks like, we also have functionality for the user to make the choice.

For the approach of finding the edges in the image, there are two main methods we chose to implement:

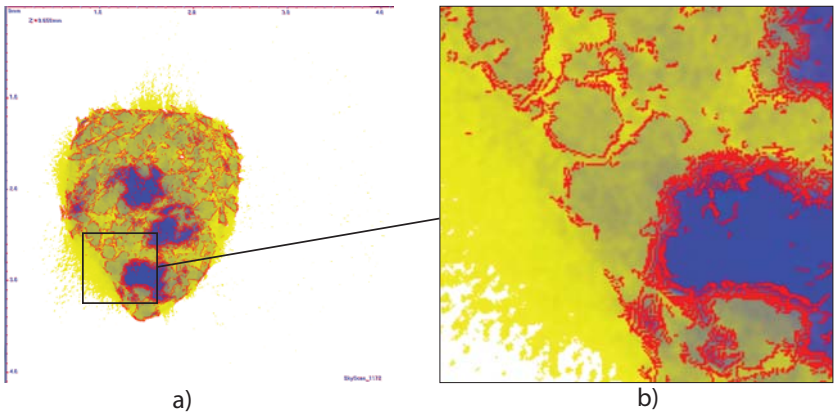


Figure 5.3: The Canny algorithm used directly on the scan-slice.

The first approach is by Canny Edge, explained in Section 4.4.2. Canny

is a method which will find edges in the center of the dataset, even if the pores in the center have a higher density than the rock on the outer parts of the dataset. This is because in the center, we will also get sharp edges between pores and rock. The negative about Canny is that it will be noisy (even if it is less noisy than Sobel Edge), because of sharpness differences in rock-density. The edges found will also, in some cases be incomplete regions, as seen in Figure 5.3b. By using a lower second threshold than used in Figure 5.3, the region could be closed, but the image will also be even more noisy. This approach is also hard to implement with the Marching Cubes algorithm, which we are using from [1].

The second approach is by using segmented regions by thresholding. This implementation needs two different flows. The first flow is when the user loads a dataset, and is illustrated in Figure 5.4. The other flow is when the user clicks or moves the mouse pointer in the image, where the flow is illustrated in Figure 5.5. Each step of the approach is explained further in the following sections. All image operations is implemented with NVIDIA CUDA, and the calculations is run by the GPU. In the entire image operations implemented, each pixels resulting color is calculated from an input image. The calculations are then independent of each others, and are then done in parallel. In an image with the width and height of 1000 pixels, there will then be launched 1 million threads to be calculated in parallel. Because of the parallelism of the image operations, the more processors on the GPU, the faster the algorithms will perform.

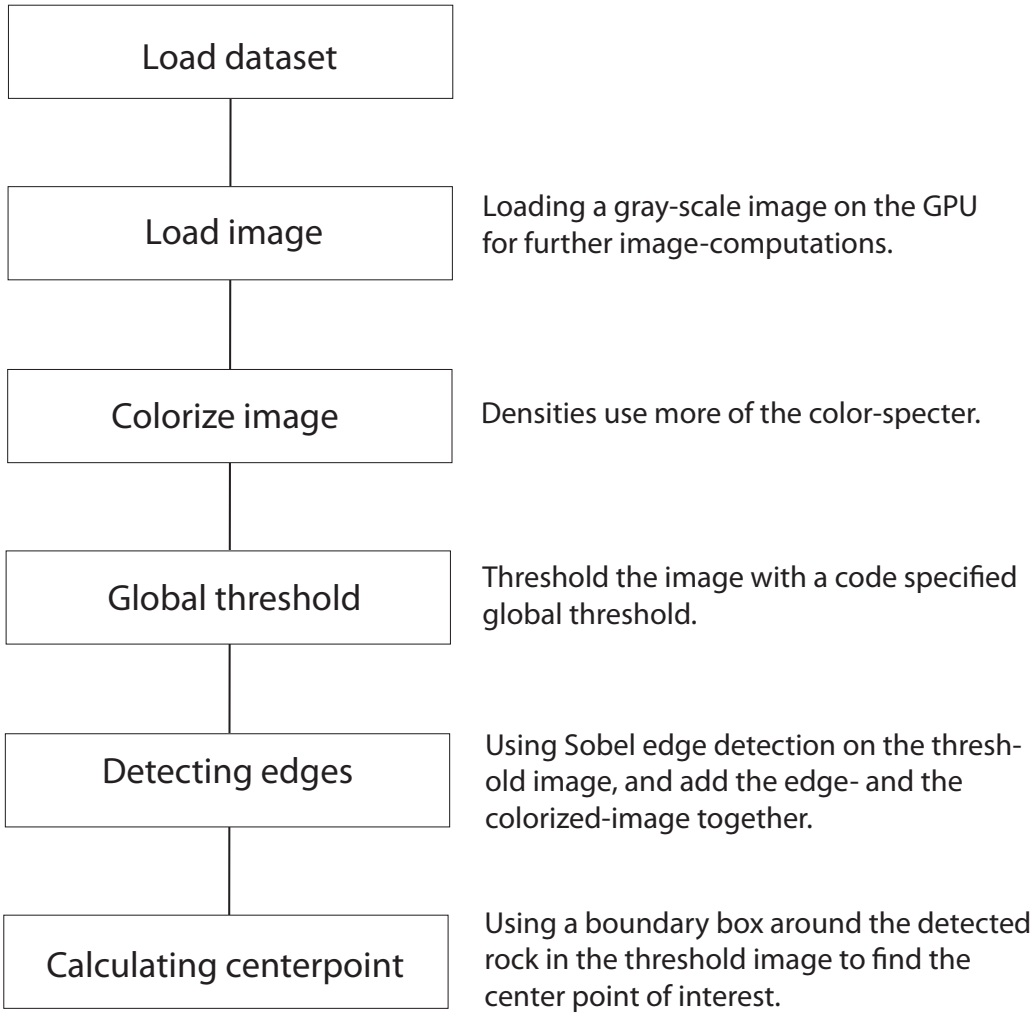


Figure 5.4: Dataset loading flow.

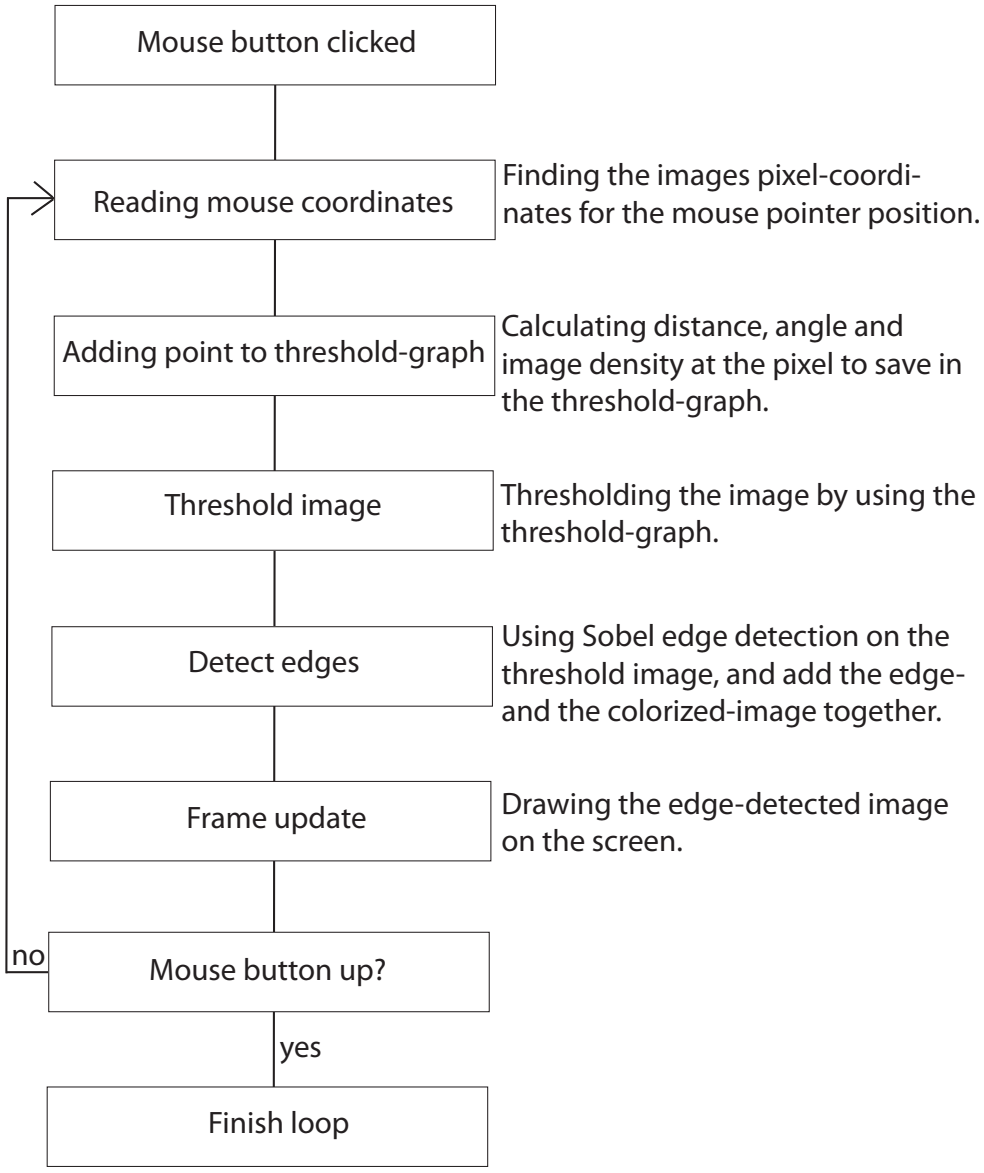


Figure 5.5: Threshold change flow.

Colorize image

It is important for the human eye to differentiate between as many densities as possible in the 2D scan data, because the user will at the end decide which parts that is rock and which parts that is pores. Then since the gray tone values can be hard to differentiate, we chose to implement a version of the image, which is using a broader part of the color specter as seen in Figure 5.6.

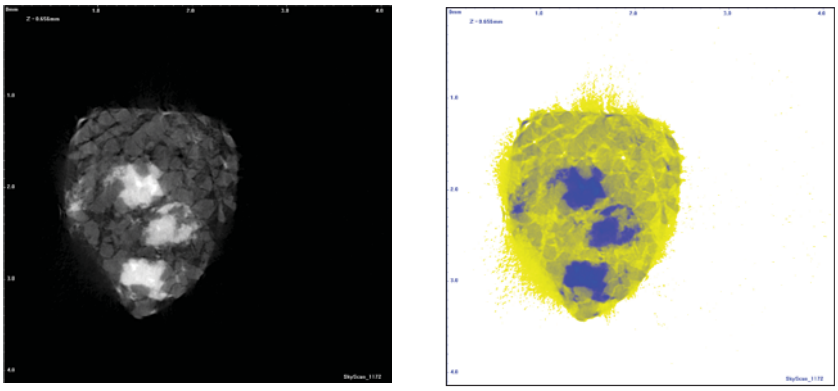


Figure 5.6: Dataset rock slice in both gray-scale and colorized mode.

To make more of the color specter available, we have used the code shown beneath. We chose to leave density values at and beneath 7 white (255,255,255). The reason of letting the lowest values be white, is because in most cases we can be sure that these values is not rock, in addition it is easier to observe the rock when the lowest values is filtered out.

```
if(pixelValue > 7)
{
    pboData[(i*4)] = 255-pixelValue;
    pboData[(i*4)+1] = 255-pixelValue;
    pboData[(i*4)+2] = pixelValue;
}
else
{
    pboData[(i*4)] = 255;
    pboData[(i*4)+1] = 255;
    pboData[(i*4)+2] = 255;
}
```

Thresholding

We are using a regional variable thresholding technique for segmentation of the image. The threshold technique is dividing the image in two component-types; what is rock and what is not rock. The two components are decided by a threshold value, which vary for each pixel and are read from a graph, which is generated from the user's mouse-clicks in the image. In our implementation, the image is divided in 8 regions as seen in Figure 5.7. For each region there will be saved a point list located in the middle of the region, which represents a graph for the threshold value. Then when the user clicks in the image with the mouse pointer, a point will be generated in the current region, and will be placed on the point list on the middle of the region. The data saved for each point is the distance from the center of the region of interest and the pixel clicked on, and the threshold value at that pixel (where the pixel will be the lowest value which is rock in that distance from the center). It is then important that the user clicks on the edges between rock and pores, on the rocks side of the edge, to get a good result.

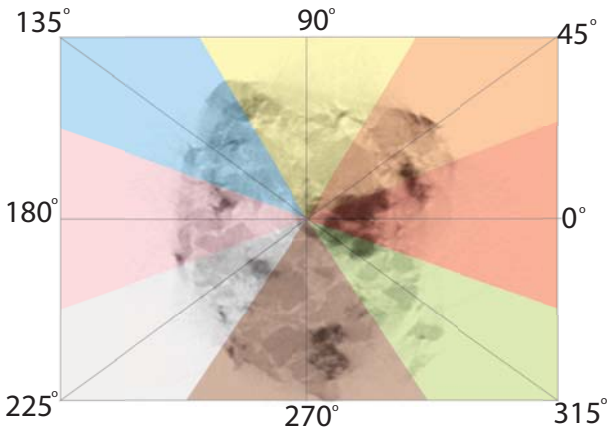


Figure 5.7: Image graph regions.

The graph made from a point list is either a straight line, with the same threshold value independent of the distance, if the point list only contains one point, as seen to in Figure 5.8a. If the point list contains two points, the graphs will be a straight line between those two, and flat before and after the two, as seen in Figure 5.8b. In a graph with more than two points, we got two possibilities; the first is by straight lines between each point, as seen in Figure 5.8c. The other method is by using a Catmul-Rom curve, which is an interpolating curve, with a smoother graph than the first possibility, as

shown in Figure 5.8d, for more information, take a look at [20]. The Catmul-Rom version requires a little more computation than the straight lines, but will generate a smoother graph.

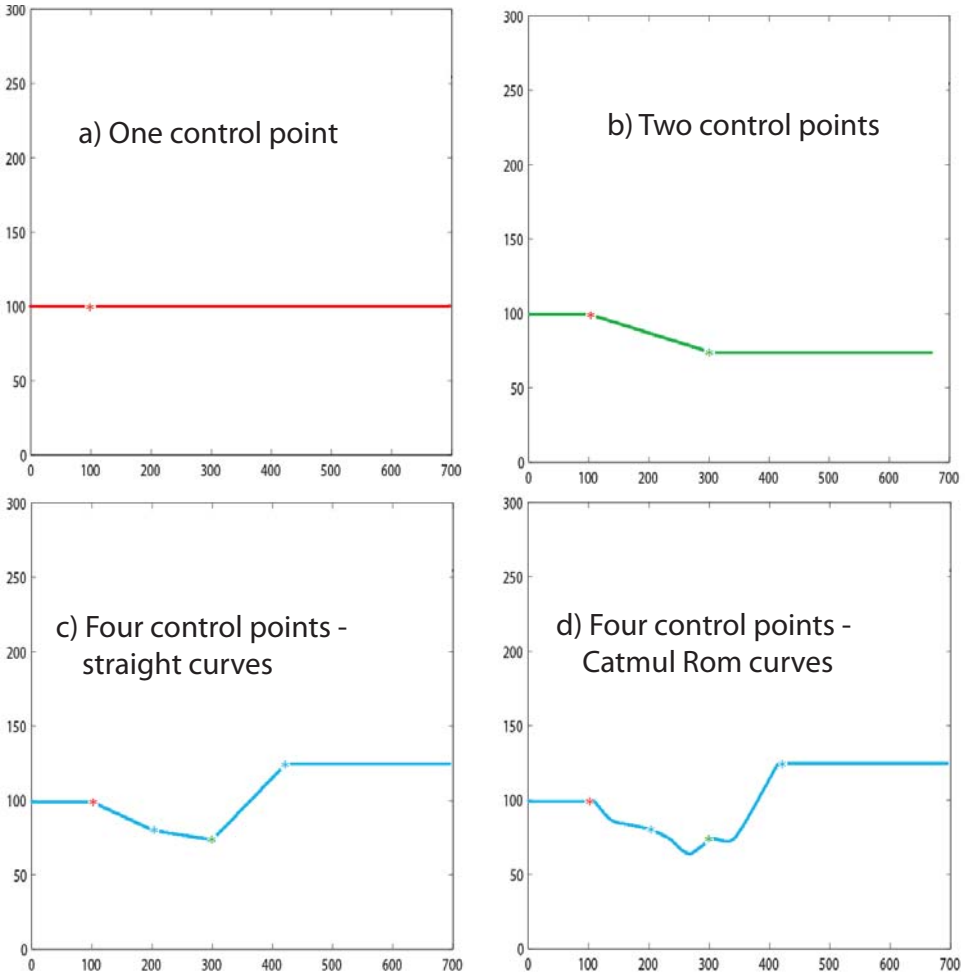


Figure 5.8: Thresholding graphs; x-axis: distance, y-axis: threshold value

Our algorithm is using the flow described in Figure 5.9 for each pixel in the current image, where each part will be described further underneath.

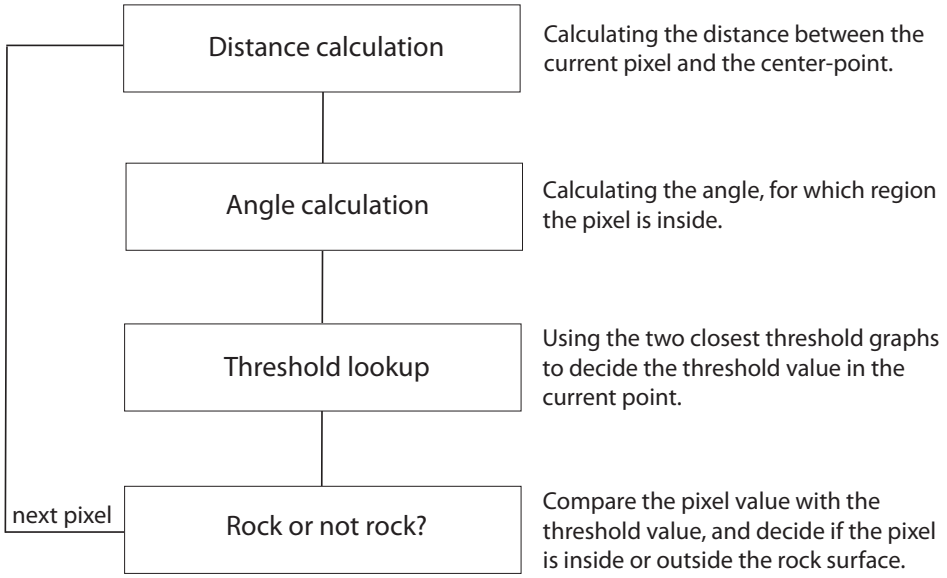


Figure 5.9: Pixel thresholding flow.

First for the current pixel, we will calculate the distance from the center point and the angle in the harmonic circle. To calculate the distance we use Pythagoras shown in Equation 4.7 and for the angle we use arctangent shown in Equation 4.9.

To look-up the threshold-values from the graphs, we are not only using the graph in the pixels region, but we are first finding the closest two point graphs with at least one point saved, as in Figure 5.10a. If no points in any of the graphs are placed yet, we will use a global threshold value. If the case is at least one point saved, the algorithm will find the closest point-graph in each direction from the current pixel. In some cases there will be only one point-graph that is populated, and the look-up will then be done in that graph. In cases where the algorithm finds different graphs in both directions, it will do a look-up in both of them. Further the algorithm will calculate the angle difference between the current pixel and each of the graphs, to use these angles to calculate an interpolation between each populated point-graph, where an example of the total graph over the whole image will look like in Figure 5.10b.

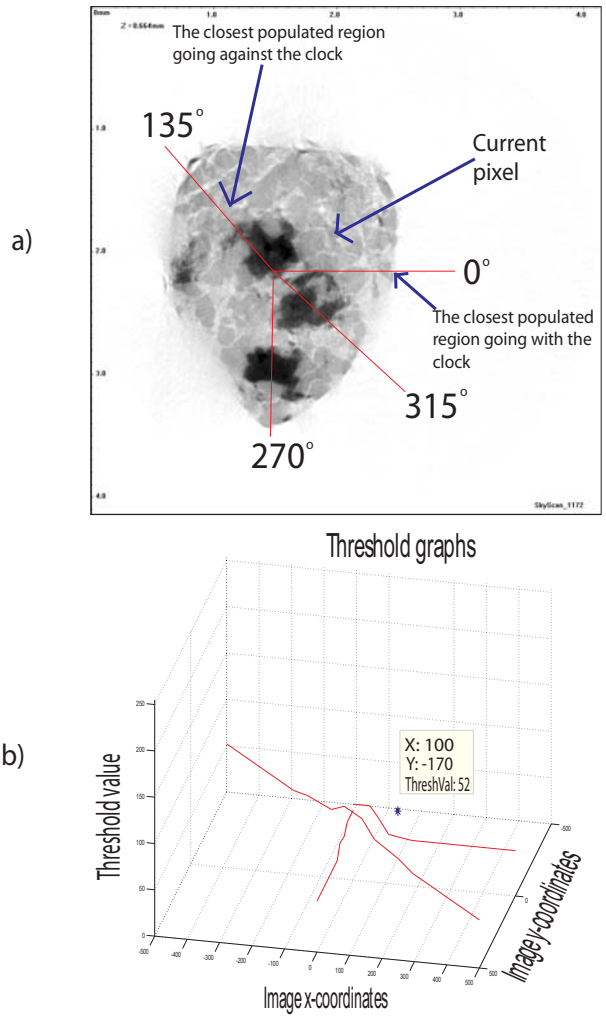


Figure 5.10: Interpolating pixel between populated regions.

Finally in the thresholding, we have calculated a threshold value, this value will then be compared against the current pixel's density value. If the pixel value is higher or equal to the threshold, the current pixel is denoted as rock, and the value is set to 255. If the case is a lower pixel value, the pixel is then not rock, and the value is set to 0.

Totally this method will be sure of getting a closed region, which is important for the 3D model and fluid simulation in it.

Detecting Edges

In this part of the algorithm, we are using the finished threshold-image to mark the edges which divides rock and pores/air. We have two different ways of marking the edges:

The first method is by using the Sobel mask over the image, for more information, take a look at Section 4.4.1. This method is a fast calculation, but will leave a thick edge around the regions.

The second method is also here by using Canny edge detection for more information, take a look at Section 4.4.2. Since there now is no noise in the image to be used, this method will give a good result, leaving a thin line around the regions. But the Canny edge algorithm is much slower than by only using Sobel.

After the edge-strength is computed by either Sobel or Canny, the edge-strengths will be compared to a threshold-value, deciding if an edge-pixel will be drawn on the screen or not. If not the pixel is an edge-pixel, it will use the colorized density-value, which will mix the most important information in both the edge-image and the density-image to a final image which the user can work with. The result of this part of the algorithm will not have any effect on the generation of the 3D volume. The algorithm is also calculated per frame when the mouse pointer is moved over the image. Therefore we chose to use the Sobel mask in this part of the algorithm.

Finding Center-Point

The center-point of the region of interest is not necessarily the center pixel of the image. In our algorithm we are interested in finding the center of the rock, and then make the regions for the threshold point-graphs in proportion to the center of the rock.

To find the center-point, we are using the global-threshold image, generated when a dataset is loaded. The first thing to do is building a bounding box around the rock in the threshold image, by finding the lowest and highest both x- and y-value for the rock in the image. After the bounding box is completed, our implementation calculates the center-point of it, as can be seen in Figure 5.11. The coordinates saved for the center-point will then be used for further calculations.

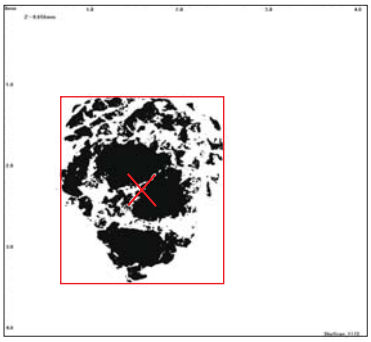


Figure 5.11: Calculation of center-point.

From the technique described over, we will not find the optimal center-point, but in most cases a center-point which is good enough for our computations to divide the rock in regions. In addition, when working with more than one image, the same center-point will be used for all the images. The reason of using the same center-point for all the images is because of only needing one distance and angle calculation per voxel in the volume rendering. The center-point will give us the ability to create good graphs in all images in the dataset.

5.3.3 Volume Rendering

For the volume rendering part of our implementation, we are using almost the same method as used in [1], but the threshold-value is calculated at the exact same method as the one shown in Section 5.3.2. While the 2D images interpolate the threshold-values from graph to graph, the volume rendering do the interpolation also in the 3rd dimention. The calculation in the 3rd dimension first finds the threshold-image which are closest to the current voxel (pixel in 3D space) in each direction. Some voxels does only have one threshold-image to calculate the threshold value from (the ones on each end of the dataset in the z-direction), and will use the same threshold lookup as the 2D images. In a voxel which are between two threshold-images, the lookup will be done on each of the images, then a calculation of the distances from the current voxel to each of the threshold-images. The closer a threshold-image is to the current voxel, the closer to the image's lookup-value the current value will be. After calculating the threshold-value, the normal Marching Cubes algorithm continues. The flow of the algorithm is shown in Figure 5.12. The 3D volume is divided in cubes, where each cube contains 8 neighbor-values in the volume and each thread operates on a different cube. For more information of the code and algorithm for marching cubes, take a look at [1].

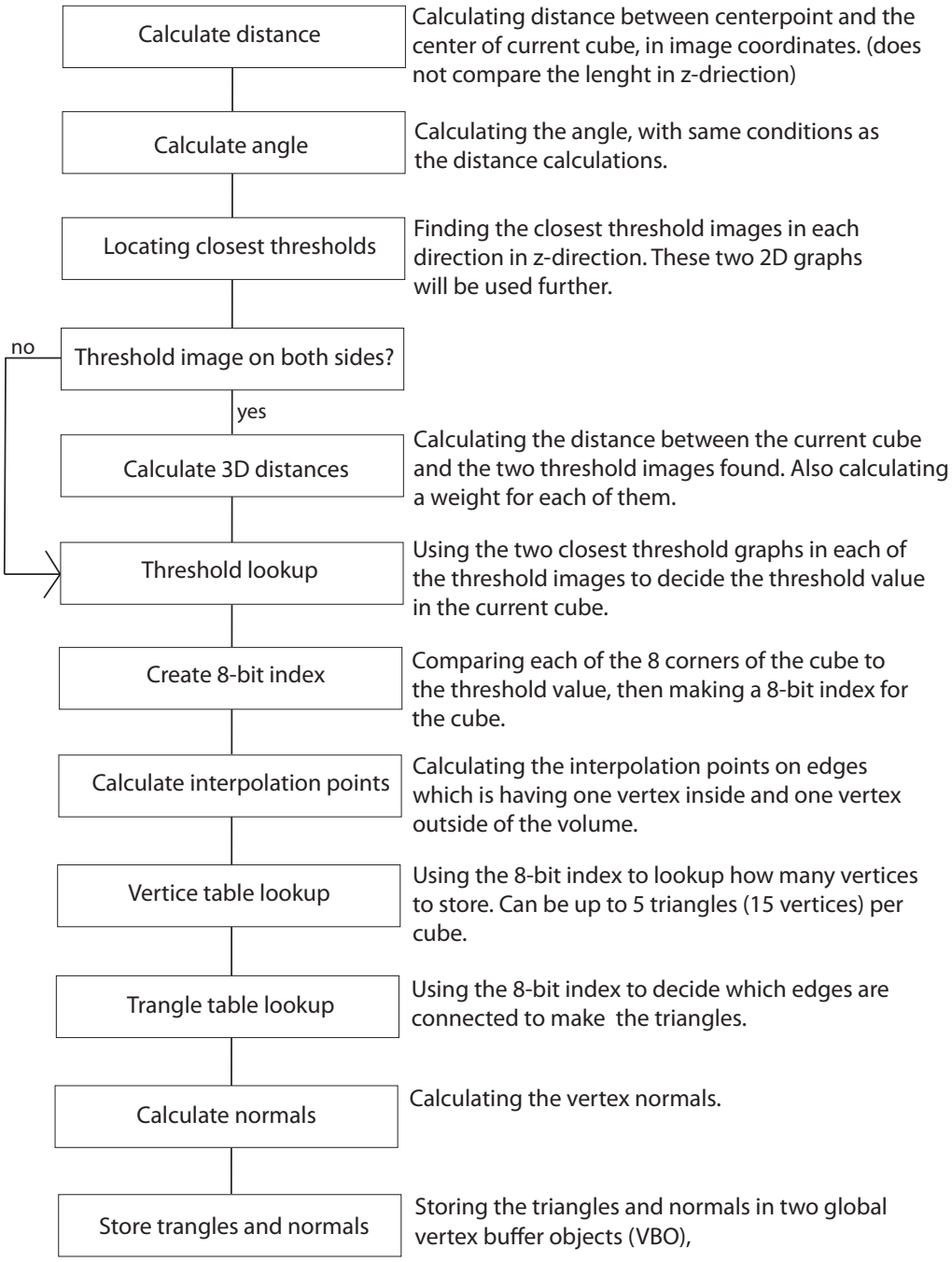


Figure 5.12: Marching Cubes, per cube flow.

5.3.4 Exporting 3D Volume to Petrel

We decided that the need of a way to save data is important in our implementation. Another important aspect is to communicate with other programs, which can use the 3D volume further for simulations or other geophysical approaches. We therefore chose to export the volume as a 3D object, which can be imported by Schlumberger Petrel.

Schlumberger Petrel supports importation of various 3D model file-formats. For our implementation, we needed a file-format which easily could be saved from the VBOs. None of the formats which saved vertices close to the same way as the VBO supported the saving of normal vectors. In addition, all of the relevant formats saved the files as ASCII.

The file format used in our implementation is the Zmap+ format for lines and points, which is saved in the way illustrated in Figure 5.13.

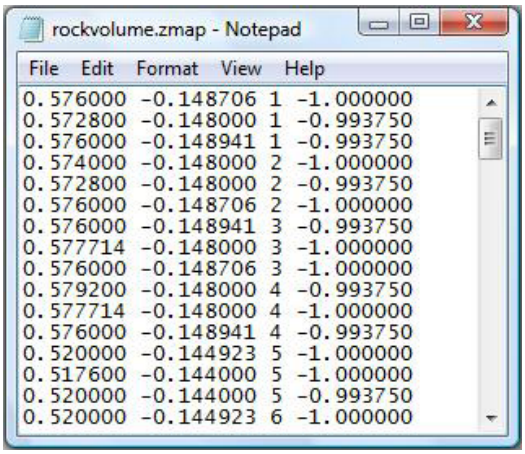


Figure 5.13: Organization of a Zmap+ file.

A Zmap+ file contains 4 columns and a row for each vertex in the 3D model. The first 2 values and the 4th in each row represent x-, y-, and z-coordinates. The 3rd column contains the polygon number, which means all vertices with the same polygon number are drawn as one polygon. Because of the file-format only save the vertices, and not their normal vectors, the inside and the outside of the rock is not differentiated in the model, as illustrated in Figure 5.14. The lack of normal vectors will also make the pores hard/impossible to observe and there is not a 3D feeling. When using full datasets, the rock will be closed, and there will be no problem doing the

simulations on the right side of the borders.

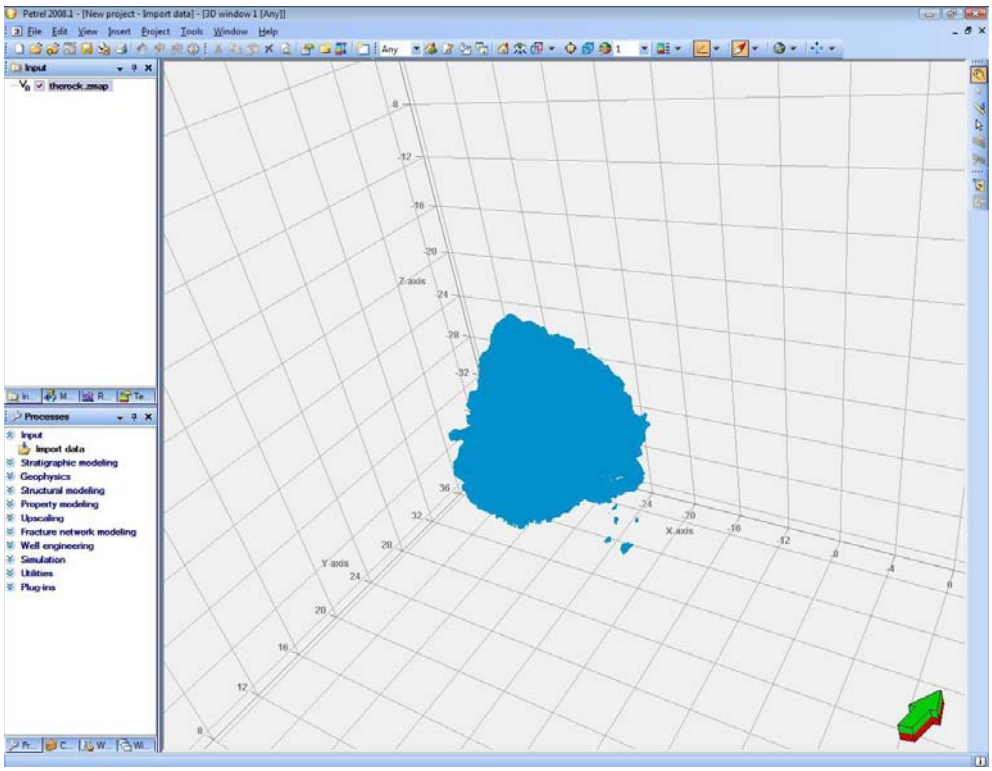


Figure 5.14: Our 3D volume loaded in Schlumberger Petrel.

Since the file-format is in ASCII, it means that the values are saved as characters, where each sign uses one byte of memory space. To save our 3D volume, there is needed to save at least 8 bytes per coordinate (float with 6 decimals) and there is 3 coordinates per vertex. In addition the polygon ID, 3 spaces and the line shift is saved per row. Totally they will use at least 30 bytes per row.

For a 3D model with 10 million vertices there is a need of at least 286.10 MB (300 000 000 bytes) memory space for storage. But for exactly that amount, every vertex has to belong to the same polygon (or up to 10 different polygons) and all coordinates has to be values between 9.999999 and -9.999999.

In our case, each dataset builds the 3D model by triangles, storing three vertices per polygon. For a dataset with 10 million vertices in our implementation, it uses about 350 MB memory space, which is not too much to save

on today's hard disks. The memory space required would be much less if we could use a binary polygon file-type instead, but the support would also have to be implemented in Schlumberger Petrel.

Our algorithm first copies the VBO from the GPU to the CPU-memory. Because of both the copying of the VBO and the preparation of each vertex for saving, the saving phase is time consuming, taking several seconds.

5.4 Memory Allocation

In our application, there is two main parts which is using the GPU's global memory. The first is digital image processing, which uses the global memory for storing images to do the computations on. The second part which uses global memory is the volume rendering part, which is very memory dependent. In addition there is used some memory on the global memory by textures and some polygons for the user interface.

5.4.1 Memory Allocated for Digital Image Processing

The digital image processing part of our implementation is not very memory dependent, compared to the volume rendering part. At all times, the original image in grayscale values and the colorized image is stored at the GPU global memory as two byte arrays. The grayscale image requires $width * height * sizeof(byte)$ bytes for storage. For the colorized image, it stores the same as the grayscale image, but 3 bytes per pixel, instead of 1. In addition to these two, there is required a PBO (Pixel Buffer Object) for saving the output, which requires a storage of 4 values per pixel. For the thresholding point graph, which also is stored in the global GPU memory, we save up to 10 reference points per graph. Having 8 graphs of 10 values in addition to a list, containing a counter pr graph; $8 * ((10 * sizeof(uint2)) + sizeof(int))$, we are using 336 bytes per threshold image. In the application, we can use up to 20 threshold-images. The only extra memory an extra threshold-images uses is the memory of a point-graph. For the per frame update we are also using an extra graph, which copies the values after the mouse button is clicked. The additional memory usage when operating with image processing techniques is a 1 byte/pixel array used for thresholding. The total memory usage of the digital image processing techniques, while computing is shown in Equation 5.1, and all time total usage is shown in Equation 5.2.

$$
\begin{aligned}
spaceallocated = \\
&(width * height * (8 + sizeof(byte)))+ \\
&(20thresholdImg * 2copies * 8graphs* \\
&((10 * sizeof(uint2)) + sizeof(int)))
\end{aligned}
\tag{5.1}
$$

$$
\begin{aligned}
spaceallocated = \\
&(width * height * (9 * sizeof(byte)))+ \\
&(20thresholdImg * 2copies * 8graphs* \\
&((10 * sizeof(uint2)) + sizeof(int)))
\end{aligned}
\tag{5.2}
$$

5.4.2 Memory Allocated for Volume Rendering

The volume rendering is very memory dependant. To control the memory, we implemented functionality that uses percentages of the total global memory on the GPU. There are created two variables to control the memory allocation. The first of these variables are verticeCount, which control how many vertices the device should maximally be able to draw. The second variable is bitmapsPrTime, where the digit represents the number of bitmaps to load to the device memory at a time.

The variable verticeCount are controlling the size to allocate for the two VBOs: posVBO and normalVBO. Each entry in each of these lists contains 4 float values, and will therefore use the allocated memory size $sizeof(float4) = 16bytes$ (pr entry in each list). The VBO posVBO is storing a vertex per entry, and normalVBO is storing each vertex's normal vector. The space allocated by the variable verticeCount is shown in Equation 5.3. For a full dataset without any compression added, the algorithm needs to store about 100 million vertices, which each has their own normal vector. For storage of 100 million vertices, we have used Equation 5.3, which will lead to: $2 * 100000000 * sizeof(float4) = 2.98$ GB. Therefore for constructing the whole data model, sometimes a compression is needed.

$$
spaceallocated = verticeCount * sizeof(float4) * 2lists \tag{5.3}
$$

A typical dataset has a size between 500 MB and 10 GB, if a user should be able both to do calculations and to save the results on the GPU, the calculations needs to be divided in some parts. The bitmapsPrTime variable controls the three main device-lists needed for computation of the Marching Cubes algorithm. The first of the device-lists is the raw data, which is stored as a byte array. The two other device-lists controls the storage-locations in the VBOs (voxel positions), where each thread in the marching cube kernel should start saving the computed vertices and normals. The size of the raw data array is $pixelsPrBitmap * bitmapsPrTime * 1byte$, and each of the two lists which are controlling the voxel positions is using double the size of the raw data, as each value is stored as a integer. Equation 5.4 shows the total space allocated by the bitmapsPrTime variable when not using compression. Equation 5.5 shows the same as Equation 5.4, but with compression.

$$\begin{aligned} spaceallocated = \\ & width * height * \\ & ((bitmapsPrTime * (sizeof(char) + (2 * sizeof(int)))) + \\ & sizeof(char)) \end{aligned} \tag{5.4}$$

$$\begin{aligned} spaceallocated = \\ & [\frac{width}{2} * \frac{height}{2} * \\ & ((\frac{bitmapsPrTime}{2} * (sizeof(char) + (2 * sizeof(int)))) + \\ & sizeof(char))] + \\ & (width * height * bitmapsPrTime * sizeof(char)) \end{aligned} \tag{5.5}$$

There are two differences between the Equation 5.5, which is using compression and Equation 5.4, which is uncompressed. The first of the differences are the halving of width, height and bitmapsPrTime. This is because the compression algorithm uses 8 points, forming a cube, at a time in the 3D

dataset, which it averages, and saves one value. The second difference is for the compression, a temporary volume of raw data is saved in full size, for doing the compression in parallel on the GPU. For an array-size, we define: $arraySize = height * width * (bitmapsPrTime + 1)$. Totally the uncompressed volume uses about 5 array-sizes for storing raw data and doing the computations with it. The first part of Equation 5.5 (the part inside the []) uses about $\frac{5}{8}$ array-sizes, while the second part of the equation uses one array-size. The total device memory used for the compressed dataset is about $\frac{13}{8}$ array-sizes.

5.4.3 Implemented Memory Allocation

In our implementation, we are using some new equations, based on the equations in Section 5.4.2. For calculating the variables bitmapsPrTime and verticeCount, we are using Equation 5.6 and Equation 5.7. Our goal was by using 75% of the total global memory on the GPU; we should be able to get a balance between the 3D volume and the performance, and in addition have enough memory to run the graphics and the digital image processing computations. We used 50% of the memory for storing the VBO, as it is important to be able to draw as many triangles as possible for storing the volume. We also use the global GPU memory size for computing both variables. The compress variable is 1 for the compressed set and 0 for the uncompressed.

$$verticeCount = (\frac{globalMemSize * (0.50 + (compress * 0.10))}{32bytes/element}) \tag{5.6}$$

$$bitmapsPrTime = \frac{\frac{globalMemSize*(0.25-(0.10*compress)}{width*height*5}}{4} * 4 \tag{5.7}$$

The variable bitmapsPrTime needs to be dividable by 4, because of the compression (uses an integer). In addition we uses 60% of the memory for storing the VBO while the compression is turned on. That is because of the each time looping through the raw data, the compressed algorithm uses double the quantity of bitmaps per time. We could improve the number of vertices to be drawn by dropping the normal vectors, but the visual rendering of our application will lose a lot of its meaning if the user is unable to observe any pores and the appearance of the rock.

Chapter 6

Benchmarks and Results

In this chapter, we will look at the performance and evaluate the visual results of our application. The first test will test the frame update rate of the image processing part. The next will compare the volume rendering part of our new threshold lookup versus the old global threshold. For evaluation of the visual results, we compare the graphical output from several different threshold methods.

6.1 Test Hardware

We are using two test-machines to be able to compare the performance on two different platforms. Therefore we have chosen to use two very different machines, performance vice. Where the NVIDIA GeForce 8800 GTS is the first cards released by NVIDIA, which has NVIDIA CUDA support. On the other computer, we have the NVIDIA Quadro FX 5800 which is a stereo rendering enabled GPU with a global memory of 4GB. Description of the two machines is seen in the Tables 6.1 and 6.2.

Table 6.1: Machine 1

CPU	Intel Core 2 Duo CPU E6600, 2.4 GHz
Memory	Corsair TWIN2X 6400 DDR2, 2 GB CL5
GPU	NVIDIA GeForce 8800 GTS No. of processor cores: 96 Processor Clock Freq: 500 MHz Memory size: 640 MB DDR3 SDRAM Memory bandwidth: 64 GB/s

Table 6.2: Machine 2

CPU	Intel Core 2 Quad CPU Q9550, 2.83 GHz
Memory	Corsair XMS3 DDR3, 8 GB
GPU	NVIDIA Quadro FX 5800
	No. of processor cores: 240
	Processor Clock Freq: 400 MHz
	Memory size: 2048 MB GDDR3 SDRAM
	Memory bandwidth: 102 GB/s

6.2 Test Data

For the tests, we are using two different datasets. The two datasets has both different size in x- and y-direction (image size), and in z-direction (numbers of images). The file types used in both datasets is 8bit BMP, containing 256 different grayscale values for density representation. These test-datasets does not contain the data of a whole scan when dealing with the performance tests. This is because of the memory restrictions of the smaller GPU, when dealing with uncompressed datasets.

The test data is generated with help from Thorvald Natvig, by a SkyScan 1137 μCT scanner in Department of Petroleum Technology, NTNU.

Dataset 1:

Dataset 1 is a scan of a porous oil rock. Where the middle image of our test-set can be seen in Figure 6.1. The middle image is the one used for the image processing functionality, when using one threshold-image. Each image in the dataset is 1000*1000 pixels, where the dataset is containing 64 images in the performance tests. For the second visual test, we are using the whole dataset, containing 570 images.

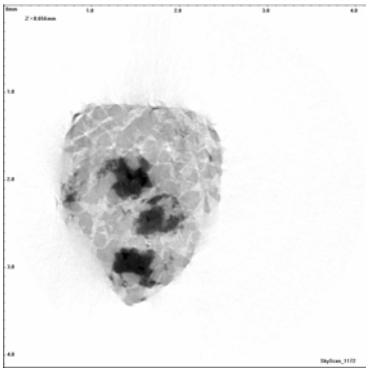


Figure 6.1: Dataset 1: Scan slice of an oil rock.

Dataset 2:

Dataset 2 is a scan of a sugar cube. Where the middle image can be seen in Figure 6.2, which also in this dataset is the one used in the image processing part of our application, when choosing one threshold-image. Each image has 1680*1680 pixels, which means almost 3 times the memory requirement as the Dataset 1's images. This test-set contains 70 images.

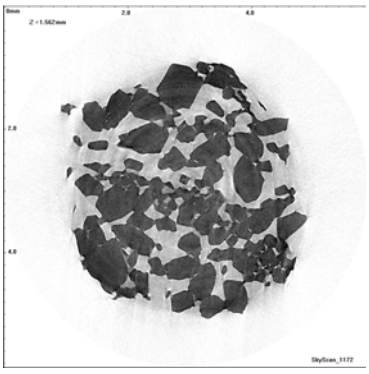


Figure 6.2: Dataset 2: Scan slice of a sugar cube.

6.3 Memory Restrictions

The detailed CT scans of the reservoir rocks used in this project generated large data sets which required a lot of memory, even when using the smaller test sets. This made it extremely important that the memory usage in our implementations was optimal, which permitted the maximum quantity of

data to be used at the same time, without running out of memory.

6.3.1 Memory used for Digital Image Processing

The image processing part is not that memory dependent. The original image in grayscale values and the colorized image are stored at the GPU global memory as two byte arrays. The grayscale image requires $1680 * 1680 * sizeof(byte)$ bytes for Dataset 2, which is about 2.69 MB. Computing the total memory restricted for the image processing part, can be done by using Equation 5.1, where the largest dataset and 20 threshold-images will be used:

$$\begin{aligned} &spaceallocated \\ &\quad = (1680 * 1680 * (8 + 1)) + (20 * 2 * 8 * ((10 * 4) + 2)) \\ &\quad = 25401600bytes + 13440bytes \\ &\quad \approx 24.24MB \end{aligned} \tag{6.1}$$

Total restricted memory for the digital image processing part is 24.24 MB. When using Machine 1, the GPUs memory is 640 MB, and the memory used for image-data will be just about 3.8%.

6.3.2 Memory used for Volume Rendering

The most memory demanding part of the implementation is the volume rendering part, which stores the vertices and normals needed for drawing the 3D volume. In addition for calculating the 3D volume, as much as possible raw data should be uploaded at a time to work with. How much data to be consumed by the volume rendering part, depends on how large 3D model we will be able to draw and how fast we will do it. The more raw data stored at a time, the faster the computations will complete.

The number of vertices to draw from a dataset is controlled by the size and the thresholding of the dataset, which is user defined. For very high or very low threshold values, either a smaller rock or fewer pores is to be drawn. In the cases of low and high thresholds, the number of vertices to be computed is reduced significantly, compared to the optimal pore models (at

least in our datasets).

For the volume rendering part of the program, the program first computes how much memory to be used for both storing raw data and for drawing the 3D volume. The amount of memory used is controlled by the variables verticeCount and bitmapsPrTime, which is explained further in Section 5.4.3. In the performance tests in this chapter, we are computing the 3D volumes without any compression. By using Equation 5.6 and Equation 5.7, we can calculate the variables, by using Dataset 2 on Machine 1:

$$\begin{aligned} &verticeCount \\ &\quad = (\frac{671088640bytes * (0.50 + (0 * 0.10))}{32bytes/element}) \\ &\quad = 10485760elements \end{aligned} \tag{6.2}$$

$$\begin{aligned} &bitmapsPrTime \\ &\quad = \frac{\frac{671088640bytes*(0.25-(0.10*0)}{1680*1680*5bytes/element}}{4} * 4 \\ &\quad = 8elements \end{aligned} \tag{6.3}$$

The total memory used for the volume rendering part is 75% of the total global memory. In the full dataset of Dataset 2, there is 1158 image slices, then for storing the whole dataset on the GPU, requires about 15.22 GB. Therefore the algorithm needs to calculate the volume in portions. And for the volume rendering part it is important to be aware of the memory management for both GPUs. Since the NVIDIA GeForce 8800 GTS card does only use 8 slices of raw data at a time for calculations, a dataset with 1158 slices will take a long time to calculate.

6.4 Performance Results

For the benchmarks, we are doing tests: The first test is a frame update test, comparing both datasets and machines. And the second is a volume rendering test, comparing our solution to the one implemented earlier in [1]. For all the performance results, we compare the two machines by the speedup factors received for Machine 2, which in all tests is faster than Machine 1.

6.4.1 Frame Updates

In this test we are comparing the FPS (Frames Per Second) gain for each of the data-sets and each of the machines.

Test description

We implemented a timer, calculating the time from one image is updated on the screen, until next image is updated on the screen. By using this time, we are also calculating the number of updates on the screen per second.

The first test is the frame update time when loading a dataset; it will also use a global threshold on the image. For more details of the flow, take a look at Section 5.3.2. The second test will compare the FPS both when placing the 1st and 10th threshold point, moving the mouse pointer for new control point calculations per frame. Each of the values is an average of 10 runs.

Results

Table 6.3: Test 1: Loading data

Dataset	Time in sec - Machine 1	Time in sec - Machine 2	Speedup
1	0.0645	0.0537	1.2011
2	0.1457	0.1323	1.1012

Table 6.4: Test 2: Frame update thresholding

Dataset	Control point #	FPS Machine 1	FPS Machine 2	Speedup
1	1	40.6100	101.8892	2.5090
1	10	27.2701	92.2034	3.3811
2	1	14.7383	54.0845	3.6697
2	10	12.4442	36.5149	2.9343

Discussion

For a visual application, at least 30 FPS is ideal for the eye to not be able to observe that a frame will stay on the screen too long, when testing different threshold values. We can observe that the NVIDIA GeForce 8800 GTS card receives close to optimal on Dataset 1, but is slowed dramatically on Dataset

2. To calculate the tenth point take more time than calculating the first. The reason of more calculation time on the 10th point, is that the algorithm has more values to compare at a time.

We can also observe that the application works ideally on the NVIDIA Quadro FX 5800, where all computations is over 30 FPS. The difference between the two GPUs is really big, where all calculations is done about 3 times as fast on the NVIDIA Quadro FX 5800. This is because it has 240 processors vs. 96 in the oldest card. There is a slower processor clock frequency on the NVIDIA Quadro FX 5800, but the global memory bandwidth is almost double the size of the NVIDIA GeForce 8800 GTS'. The global memory bandwidth is giving such a big part of the speedup because of each thread is operating with its own pixel, and each pixel value in the output image in some cases are computed from up to 25 pixels in the input image. The need of using the global memory is then a large part of each thread's computation time.

6.4.2 Volume Rendering

In this test we are comparing the old versus the new version of the volume rendering algorithm.

Test description

As mentioned before, we implemented the Marching Cubes algorithm in [1], which was accelerated for GPU calculations. Our implementation is based on that earlier approach. The main difference between our old and new algorithm is how we do the threshold value lookup. In this test we compare the calculation time of the old Global Threshold (GT) functionality versus the new Variable Regional Threshold (VRT). The test is to be done on each of the datasets on each of the computers. We will also test the difference of computation time between a threshold with one region populated and a threshold with all eight regions populated.

In addition, all the VRT tests also will be tested with both a dataset with one threshold-image and a dataset with 10 threshold-images.

Results

Table 6.5: Test 3: Volume rendering computation

Dataset	Threshold type	Regions	Threshold images	Time in sec Machine 1	Time in sec Machine 2	Speedup
1	VRT	1	1	1.6772	0.4240	3.9557
1	VRT	8	1	2.1544	0.4651	4.6321
1	VRT	1	10	2.4287	0.9280	2.6171
1	VRT	8	10	2.8939	0.9595	3.0159
1	GT	N/A	N/A	0.6968	0.1573	4.4298
2	VRT	1	1	4.7581	1.2213	3.8959
2	VRT	8	1	5.8968	1.2820	4.5997
2	VRT	1	10	7.4276	2.5891	2.8688
2	VRT	8	10	9.3919	2.7232	3.4488
2	GT	N/A	N/A	2.4777	0.7275	3.4058

Discussion

In the volume rendering part of the implementation, we are doing some of the same computations two times per pixel. First we are running half of the marching cubes algorithm (doing the first 7 and the 9th step in Figure 5.12), to calculate where in the VBOs to save the different cubes' vertices and normals. Then the whole flow seen in Figure 5.12 is done. In both of these parts, the new code from this thesis is executed for finding each cube's threshold value. We can observe that the threshold lookups is time-consuming in proportion to the rest of the algorithm, which is the difference between GT and VRT.

We can also observe that the more regions populated, will receive larger computation time. This is because the region lookup calculation is done only once for each pixel in the one regioned computation. In the computations using one threshold-image, there is one image threshold lookup on every voxel in the volume. When using ten threshold-images, the most of the voxels will need two lookups for deciding the threshold value. This is because our implementation interpolates between the threshold-images, and will result in a higher computation time for using ten threshold-images.

The speedup on the volume rendering test is even greater for the NVIDIA Quadro FX 5800 in some of the cases, than the speedup in the image op-

erational test. In addition to the elements like number of processors and memory bandwidth, the NVIDIA Quadro FX 5800 card has a 4GB global memory, which means that much more raw data can be uploaded at a time than on the NVIDIA GeForce 8800 GTS with 640MB global memory. This also means that each computation in the flow can be done a higher number of times, before another computation is calculating further.

The speedup gained when using one threshold-image is greater than when using ten. The main difference between one and ten threshold-images computationally, is that the regional lookup is being done two times for most threads instead of one. Therefore there more computations is executed per thread when using the ten threshold-images. The NVIDIA GeForce 8800 GTS' processors is using a higher frequency for computations, which means more computations per thread will speedup the thread in proportion to the NVIDIA Quadro FX 5800 (more computations per thread, the lesser speedup). Using eight populated regions gets more speedup than when using one populated region. When using eight regions, there is used much more global memory lookups than when using one region. Since the NVIDIA Quadro FX 5800 has a higher memory bandwidth than the NVIDIA GeForce 8800 GTS, it receives a higher speedup by using 8 populated regions.

6.5 Visual Results

In this thesis, the main focus is on the visual results. In this section we are comparing the visual results gain from our implementation.

6.5.1 Comparing 2D Thresholding

In this test we are comparing the 2D digital image processing results.

Test description

We have implemented some different ways to differentiate the rock and pores via thresholding. In this test we will compare the results from a Global Threshold (GT), a Variable Threshold (VT) and a Variable Regional Threshold (VRT). This test will be done on Dataset 1, since the sugar cube is relatively uniform, and will give a good result even with a global threshold.

Results

First we will show the results of the three different thresholding techniques in Figure 6.3. Then in Figures 6.4, 6.5, and 6.6 there will be some zoomed images, for easier observation of the differences. And finally in Figure 6.7 we will compare the 3D models from each of the results.

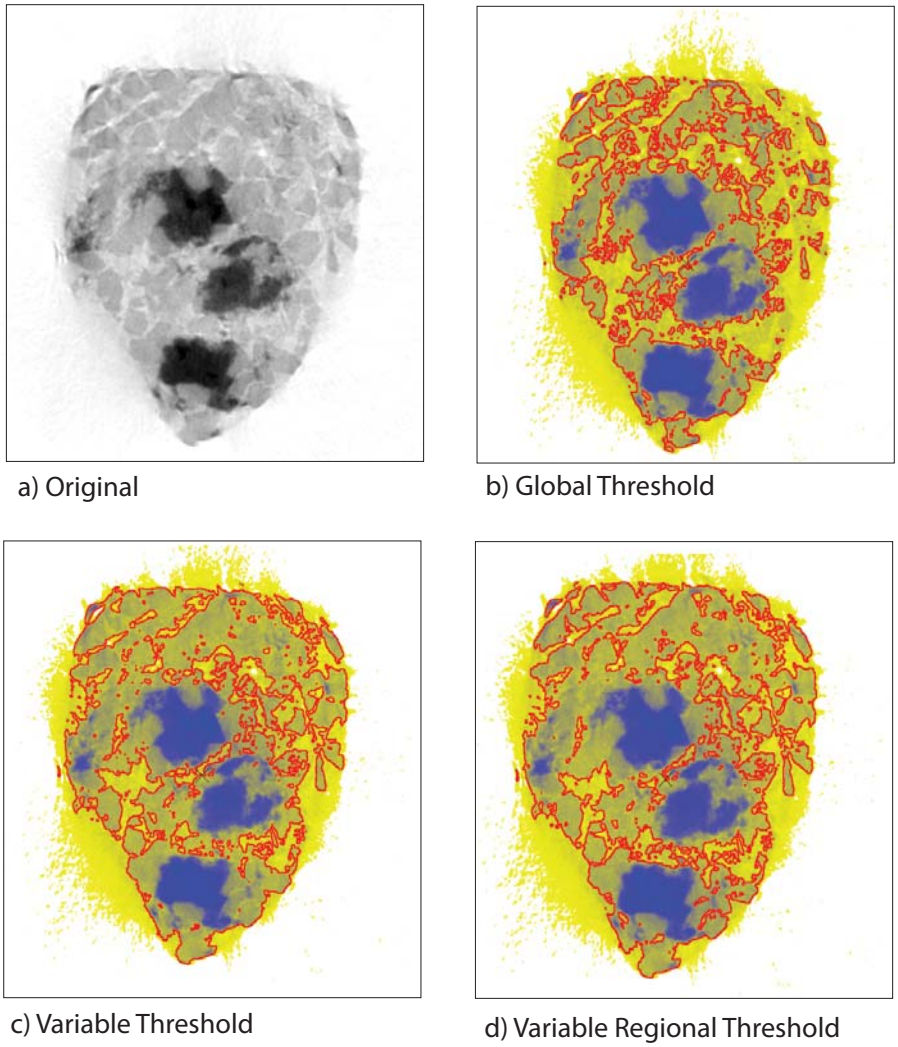


Figure 6.3: Thresholding results.

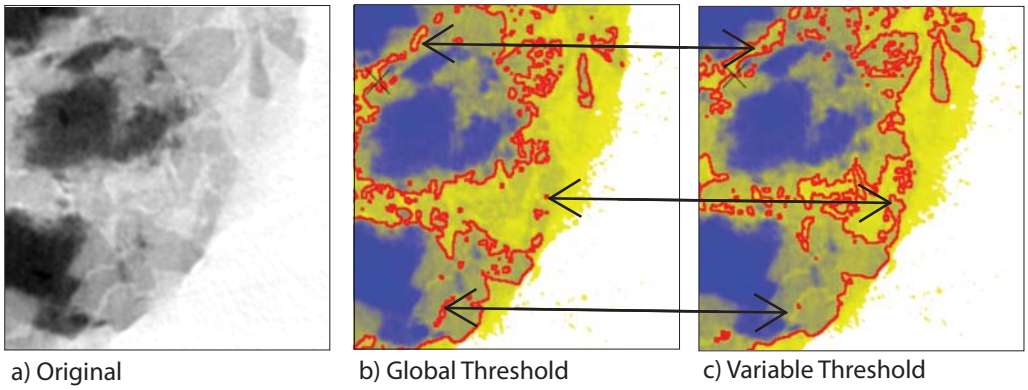


Figure 6.4: Global Thresholding VS. Variable Thresholding.

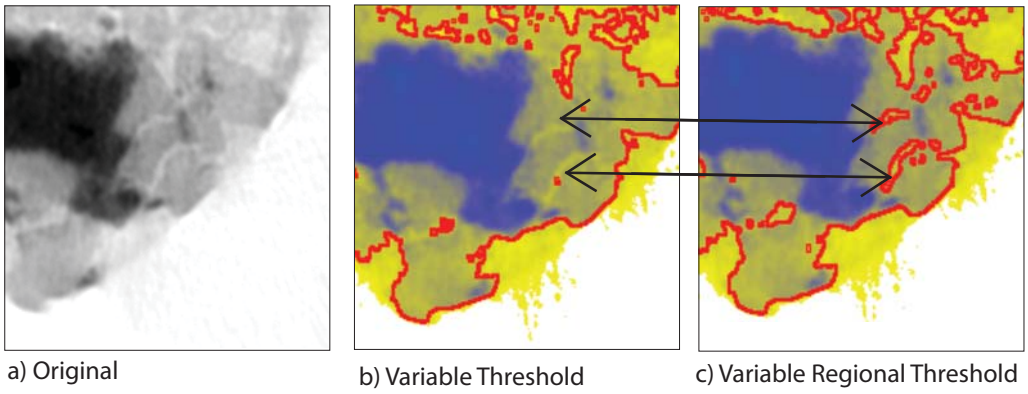


Figure 6.5: Variable Thresholding VS. Variable Regional Thresholding.

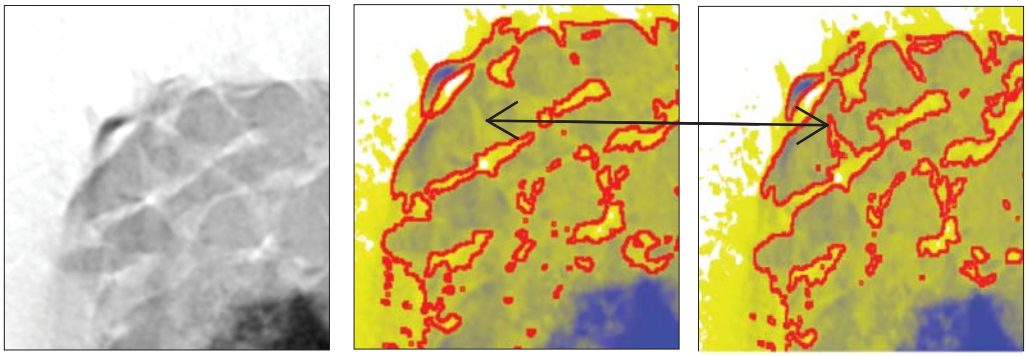


Figure 6.6: Variable Thresholding VS. Variable Regional Thresholding.

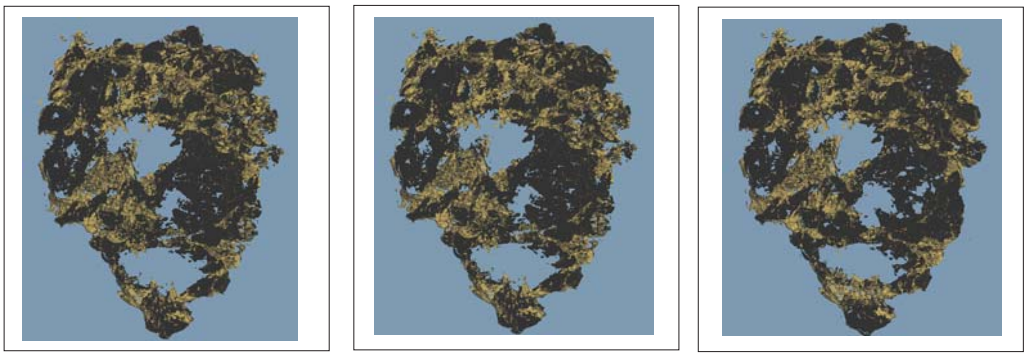


Figure 6.7: 3D volumes of the 3 thresholding methods.

Discussion

In Figure 6.3, the different results are presented. From these images we can observe that the more complicate functionality in our implementation is closer to finding the edges in the original than the simpler methods.

A comparison between GT and VT can be seen in Figure 6.4. We can observe that the GT finds the rock some places, but both miss some rock and some pores, which means that a GT will not give an optimal result. On VT, we have marked some of the rock which GT did not find, but as a result some parts of the pores which the global threshold found were not found by the VT.

In Figures 6.5, and 6.6, we are comparing VT and VRT. In both figures we can observe some pores VRT found, which VT didn't. The reason why VRT is closer to the original rock, is that the user is able to adjust the borders on all sides of the image.

6.5.2 Comparing 3D Volume

In this test we are comparing the 3D volume results gained from using 1 and 10 threshold-images.

Test description

In our implementation, we can choose to operate on between one to ten images in the scan data. By operating with more than one image, the program will interpolate the threshold values between the threshold images. In this test, we will compare the 3D volumes generated from using one and ten threshold-images. This test will be done on the full Dataset 1, with 570 scan data slices. By using one threshold-image, the 285th image will be up for our digital image processing techniques. Therefore we will compare how image no. 50 will be thresholded, when adjusting the edges to the 285th image, vs. how it will be when using a much closer adjusted threshold.

Results

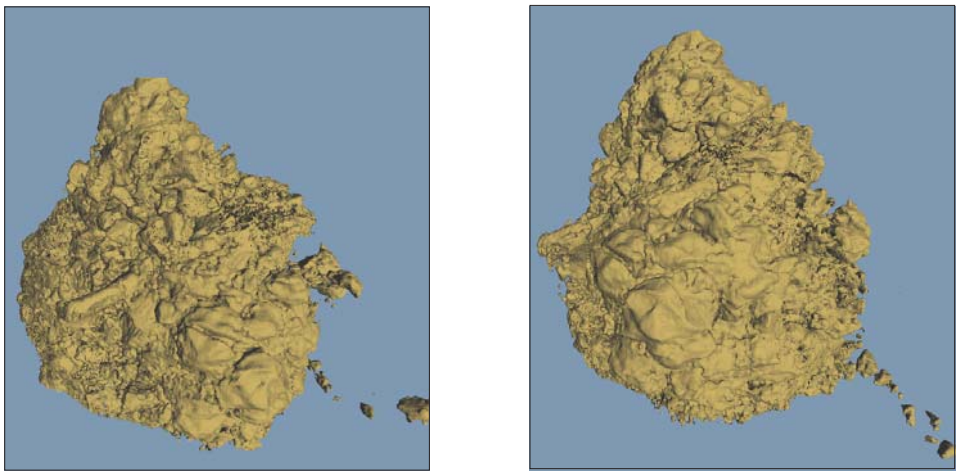


Figure 6.8: 3D volume using 1 image slice, shown from different directions.

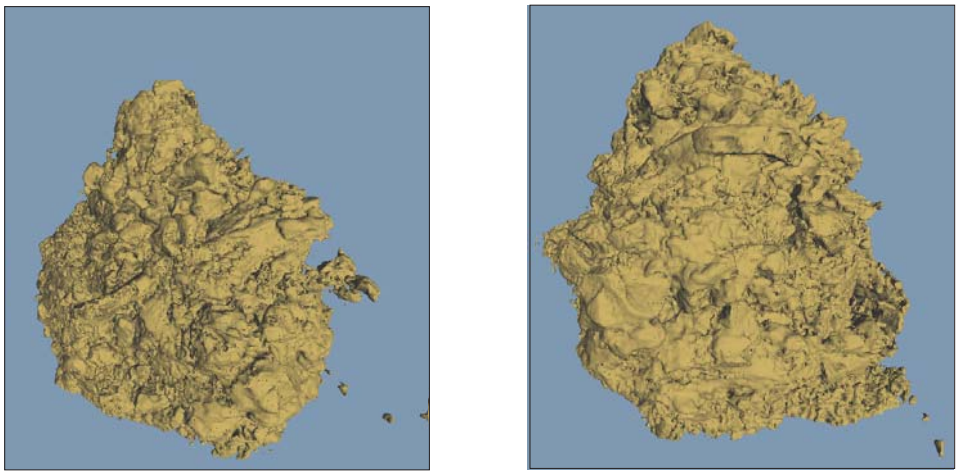


Figure 6.9: 3D volume using 10 image slice, shown from different directions.

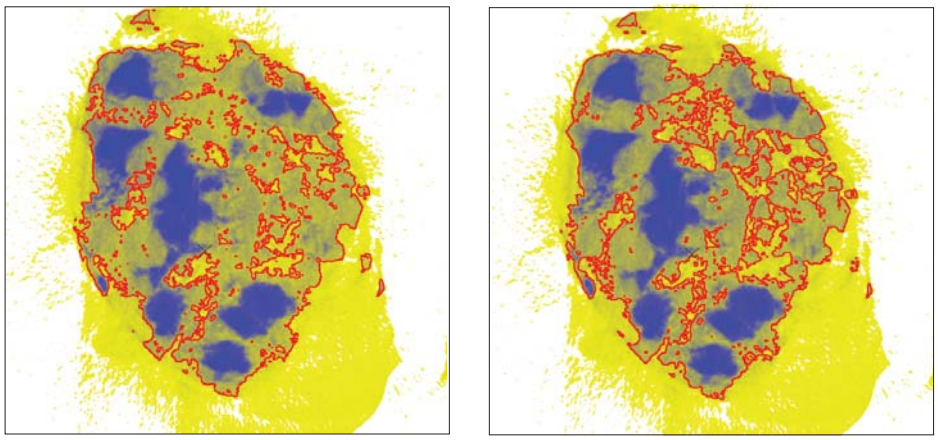


Figure 6.10: Image #50 thresholded with 1 and 10 slices.

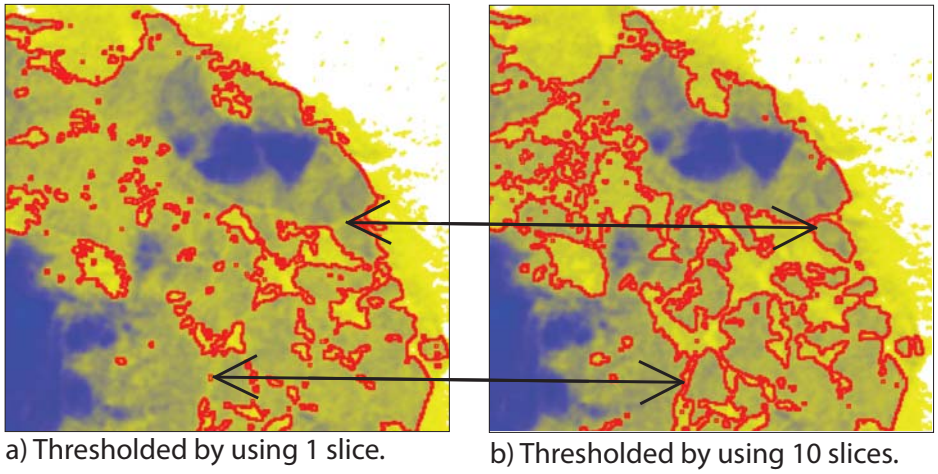


Figure 6.11: Image #50 thresholded with 1 and 10 slices - differences.

Discussion

In Figures 6.10 and 6.11, we can observe the threshold generated in image #50 in the dataset by using both one and ten image slices, and that there is some differences in the threshold adjusted for the middle image and the 50th image. When doing the operations on the middle image of the dataset only, the distance between the image operated on and image #50 is 235 images. Over 235 images, the variation in the image can be the same as the difference in 235 pixels in an image, but the hardening artifact will not be as significant in the z-direction as in the two other directions. Therefore the threshold-graph generated by the middle image, will often not suit the images on the top or bottom of the dataset.

As was seen in Section 6.4.2, the volume uses much more computation time when interpolating threshold-images, than using the one in the middle. In this visual test, we can conclude that the visual result of using interpolating threshold-images will create a 3D volume closer to the real rock. The visual results are also the main focus in our thesis, which is the reason why we chose to be able to use several slices in the final version.

Chapter 7

Conclusions and Future Work

Today's GPUs (Graphical Processing Units) are not only used for doing the graphics rendering in computer games, but can with their several hundred computational cores also be attractive computational platforms for parallelized applications which earlier needed the computational power of large CPU clusters. GPUs also open up for many real-time applications that previously were considered too computationally intensive to do.

7.1 Conclusion

In the oil business, it is important to get as detailed and realistic 3D models of porous rocks as possible to be able to predict the flow of fluids through the rocks. Unfortunately, different artifacts, especially hardening artifacts, makes the CT scan data inexact. Earlier methods used the scan data directly to make a 3D model based on a global threshold (border between rock and pores). Because of artifacts in the CT scan, the earlier 3D models also became inexact. In this thesis, we showed how the computational power of the GPU could be used to develop a tool for interactively enhancing the CT scan data of porous rocks.

The application we developed included image processing techniques for differentiating between rock and pores in the scan-data of core samples. The results of these techniques are further used to generate 3D models based on the CT scans of core samples. Our implementation can handle large datasets of at least 1000 x 1000 x 570, in order to generate detailed and realistic models. We achieved this by letting the bulk of the calculations be handled by GPUs. Our approach gave the user the ability to interactively improve core-sample data by letting the user move estimated pore outlines, in order

to make a better 3D model than earlier approaches.

We tested our application on both an older GPU and a newer high-end graphics card. On the latter, our implementation performed very well achieving frame updates over the ideal 30 FPS allowing the user to fully interactively control the pore borders.

We also implemented a method to export the 3D models made in our implementation to the Schlumberger Petrel framework. This export of the 3D models opens the possibility to use our implementation for further use in a geophysical application.

Other applications relying on scanned data that wishes to differentiate between to materials or a material and a void, could also benefit from our approach (e.g. in medical imaging). However, this was not explored in this thesis.

7.2 Future work

There are a number of ways that our implementation could be improved, but in most cases with a tradeoff.

2D image operations

The edge detection algorithm used in our final implementation, to show the border between rock and pores in the 2D image display, draws a rather thick contour. The real computations executed while generating the 3D volume use no more than 1 pixel thick edges. Since the thick contour is drawn on the 2D display, the user can be confused and choose a slightly wrong value for the edges. This can be the reason why the 3D volume gets a little inexact. We also implemented the Canny edge detector for drawing the edges, but since the program requires a good frame rate, we discarded Canny in the final version. A way to resolve this is by using a edge thinning algorithm on our solution. The edge thinning will decrease the performance, but on the newer and future GPUs it could be a function to turn on.

A way to increase the accuracy of the edge finding is by region-based segmentation techniques, like region growing or watershed. These techniques can find the image centers affected by the hardening artifacts. After the cen-

ters and the pixels are sorted in regions by which center they are attached o, the use our thresholding technique in each of the regions can be done. This technique will open a possibility to more accuracy in the images the technique operates, but will require more computations.

When doing image operations on the GPU, the image-data is read from the global memory. Since the images are stored in the memory as byte arrays, we are using 1 byte transfers. The GPU's bandwidth is specialized for transfers of float values, which is 4 bytes. Therefore we can receive a speedup by reading 4 bytes at a time, and then store the values which are to be used by more than one thread in the thread-group in the shared memory. By saving the values on the shared memory, the image mask computations, like sobel will receive a speedup.

3D volume rendering

The volume rendering algorithm, Marching Cubes (MC) , is slowed down dramatically by using the threshold lookup up to four times per cube (both when counting and saving the vertices, and for each of the nearby threshold images). In our preliminary work [1], we introduced a way to implement histopyramids, which is a reduction algorithm, to accelerate our MC algorithm.

Another way to do the 3D rendering is by using a direct approach, like shear-warp. This then will calculate the outside of the volume per frame, instead of creating a polygon structure. A direct rendering algorithm also opens a possibility to threshold the whole volume, slice by slice, saving the rock and pores as boolean data. The dataset can then use full resolution, without using too much storage space. The drawback with using a direct rendering technique is the required computations per frame.

File support

The implementation currently supports only datasets of 8 bit BMP files for input and Zmap+ as output. For making the application more useable with support for more geophysicists, there should also be implemented support for various types of other file formats.

One of the file formats that should be implemented as output is VTK, which is supported by Eirik Ola Aksnes' lattice Boltzmann GPU implemen-

tation. The 3D models can then be used further for simulations of fluid.

Bibliography

[1] Eirik Ola Aksnes and Henrik Hesland. Gpu techniques for porous rock visualization, 2008. Project work in TDT4590 Complex Computer Systems, Specialization Project in Norwegian University of Science and Technology. http://www.idi.ntnu.no/ elster/master-studs/aksnes-hesland-MSproj.pdf.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley, 2006. Electrical Engineering and Computer Sciences University of California at Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[3] Maria Axelsson, Stina Svesson, and Gunilla Bergfors. Reduction of ring artifacts in high resolution x-ray microtomography images, 2006. Centre for Image Analysis, Swedish University of Agricultural Sciences. http://www.springerlink.com/content/dk42mr42337381j6/fulltext.pdf.

[4] Julia F. Barrett and Nicholas Keat. Artifacts in ct: Recognition and avoidance, 2003. http://www.labmeeting.com/paper/25821798/barrett-keat-2004-artifacts-in-ct-recognition-and-avoidance.

[5] David Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5), 2008.

[6] John F. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1986.

[7] Ignacio Castaño. 10 fun things to do with tessellation, 2009. http://castano.ludicon.com/blog/2009/01/10/10-fun-things-to-do-with-tessellation/.

[8] Dark CodeX. Amplexus primordia, 2009. http://pouet.net/prod.php?which=52934.

[9] M. E. Coles, E. L. Muegge, and E. S. Sprunt. Applications of cat scanning for oil and gas production research. *IEEE Transactions on nuclear science*, 38(2), 1991.

[10] NVIDIA corporation. *NVIDIA Cuda Compute Unified Device Architecture Programming guide*. NVIDIA corporation, 2.0 edition, 2008. available from http://developer.download.nvidia.com/compute/cuda/20/docs/NVIDIA CUDA Programming Guide 2.0.pdf.

[11] Ian A. Cunningham and Philip F. Judy. Computed tomography, 2000. CRC Press LLC. http://www.sovem.org.ve/biblioteca/Computed%20Tomography.pdf.

[12] Christopher Dyken and Gernot Ziegler. High-speed marching cubes using histogram pyramids. *EUROGRAPHICS 2007*, 26(3), 2005. http://www.mpi-inf.mpg.de/g̃ziegler/hpmarcher/hpmarcher.pdf.

[13] E. Elsen, M. Houston V. Vishal, P. Hanrahan V. Pande, and E. Darve (Stanford University). N-body simulations on gpus. Available: http://www.arxiv.org/abs/0706.3060.

[14] Laura A. Freberg. *Discovering biological psychology*. Houghton Mifflin, 2006.

[15] Kevin Gee. Introducing directx 11, 2008. http://www.gamasutra.com/view/feature/3759/sponsored feature introducing .php?print=1.

[16] Ryan Geiss. Generating complex procedural terrains using the gpu. *GPU GEMS 3*, 2007.

[17] Frank Goetz, Theodor Junklewitz, and Gitta Domik. Real-time marching cubes on the vertex shader. *EUROGRAPHICS 2005*, 2005. http://www.cs.uni-paderborn.de/fileadmin/Informatik/AG-Domik/medicine/ Real-Time_Marching_Cubes_on_the_Vertex_Shader___EG_2005_.pdf.

[18] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson Prentice Hall, 3rd edition, 2008.

[19] Bill Green. Canny edge detection tutorial, 2002. http://www.pages.drexel.edu/w̃eg22/can tut.html.

[20] Donald Hearn and M. Pauline Baker. *Computer graphics with OpenGL*. Pearson Prentice Hall, 3rd edition, 2004.

[21] Andreas Kayser, Mark Knackstedt, and Murtaza Ziauddin. A closer look at pore geometry, 2006. Schlumberger Oilfield Review. http://www.slb.com/media/services/resources/oilfieldreview /ors06/spr06/01_pore_geometry.pdf.

[22] Andreas Kayser, Andrew Curtis Rutger Gras, and Rachel Wood. Visualizing internal rock structures, 2004. http://www.slb.com/media/services/resources/articles/software/offshore _introckstruct.pdf.

[23] Leif Christian Larsen. Framework for polygonal structures, computations on clusters. Master's thesis, Norwegian University of Science and Technology, 2007. http://daim.idi.ntnu.no/masteroppgaver /IME/IDI/2007/3499/masteroppgave.pdf.

[24] John L. Manferdelli, Naga K. Govindaraju, and Chris Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5), 2008.

[25] Michael D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5), 2008.

[26] Jason L. Mitchell, Marwan Y. Ansari, and Evan Hart. Advanced image processing with directx 9 pixel shaders. *ShaderX 2*, 2003. http://ati.amd.com/developer/shaderx/ShaderX2 _AdvancedImageProcessing.pdf.

[27] Jun-Taek Oh and Wook-Hyun Kim. Ewfcm algorithm and region-based multi-level thresholding. *School of EECS*, 2006. http://www.springerlink.com/content/c56027561h2up130/fulltext.pdf.

[28] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, James, and C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5), 2008.

[29] Suryakant Patidar, Shiben Bhattacharjee, Jag Mohan Singh, and P. J. Narayanan. Exploiting the shader model 4.0 architecture, 2004. Center for Visual Information Technology, IIIT Hyderabad. http://research.iiit.ac.in/s̃hiben/docs/SM4_Skp-Shiben-Jag-PJN_draft.pdf.

[30] C. Raven. Numerical removal of ring artifacts in microtomography, 1998. http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=RSINAK000069000008002978000001&idtype=cvips&gifs=yes.

[31] Charles Loop (Microsoft Research) and Scott Schaefer (Texas A&M University). Approximating catmull-clark subdivision surfaces with bicubic patches, 2007. http://research.microsoft.com/en-us/um/people/cloop/acctog.pdf.

[32] Christopher I. Rodrigues, David B. Kirk, Sam S. Stone, Sara S. Baghsorkhi, Shane Ryoo, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008. http://grothoff.org/christian/teaching/2008/4704/p73-ryoo.pdf.

[33] Jürgen Peter Schulze-Döbold. Interactive volume rendering in virtual environments. Master's thesis, Universität Stuttgart, 2003. http://elib.uni-stuttgart.de/opus/volltexte/2003/1466/pdf/schulze.pdf.

[34] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc, 2.0 edition, 2009. http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf.

[35] Jan Sijbers and Andrei Postnov. Reduction of ring artifacts in high resolution micro-ct reconstructions. *Phys. Med. Biol.*, 49, 2004.

[36] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. Thomason, 3rd edition, 2008.

[37] Haining Sun, Shaokun Qiu, Shanshan Lou, Jinjun Liu, Changjun Li, and Genmiao Jiang. A correction method for nonlinear artifacts in ct imaging. *Proceedings of the 2004 IEEE*, 2004.

[38] Yi Sun, Ying Hou, and Jiasheng Hu. Reduction of artifacts induced by misaligned geometry in cone-beam ct. *IEEE Transactions on biomedical engeneering*, 54(1), 2007.

[39] David Tarjan and Kevin Skadron. Multithreading vs. streaming. *MSPC08*, 2008. http://www.cs.virginia.edu/s̃kadron/Papers/mspc08_final_tarjan_skadron.pdf.

[40] Edward R. Dougherty (Texas AM University) and Junior Barrera (University of Sao Paulo). Logical image operators. In *Nonlinear Filters for Image Processing*, 1999.

[41] Kezhou Wang, Thomas s. Denney Jr., Edward E. Morrison, and Vitaly J. Vodyanoy. Construction of volume meshes from computed tomography data. *Proceedings of the 2005 IEEE*, 2005.

[42] Wikipedia. Instruction level parallelism, March 2009. Available: http://en.wikipedia.org/wiki/Instruction level parallelism.

[43] Wikipedia. Volume rendering, March 2009. Available: http://en.wikipedia.org/wiki/Volume rendering.

[44] Barry Wilkinson and Michael Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Prentice Hall, 2nd edition, 2005.

Appendix A

Source Code Overview

In this appendix, we give an overview of the architectural structure of our implementation.

A.1 File Overview

The program is written in C/C++ in a Visual Studio 2005 project. For the computational part, there is used NVIDIA CUDA, and for the graphical functionality OpenGL is used. The individual functions in the source files are commented, so in this appendix we will only give a file overview. The cpp files does not include any of the GPU functions, they do call them from the cu file.

Our implementation consists the following files:

- *main.cpp:* Contains all the functionality for initiation and rendering of the OpenGL (except for the volume rendering), button and mouse events, binding of thresholded images to textures, zooming on the 2D image, adding of points to the point graph (reading mouse coordinates and calling a function from the CUDA-file), memory allocation calculations, and functionality for the timers.
- *bmpload.cpp:* Handles loading of a dataset as a set of 8bit BMP-files. It includes functions for finding the number of files in the folder and calculating the size of the images. In addition, it also includes the functionality for loading the bitmap to work on with the digital image processing functionality. The last main functionality is for loading a part of the dataset up to the GPU, needed for the marching cubes computations.

- *imageTech.cpp:* Handles all the functionality for the digital image processing. It includes functionality for making the colorized image of the original, in addition to two large functions. The first of the two large functions is the one for thresholding the bitmap, which includes the last 3 steps of Figure 5.4, and is explained further in Section 5.3.2. The other large function is the Canny Edge algorithm which is not used in the final release, but is there in case of further development, and is explained further in Section 4.4.2.

- *marchingCubes.cpp:* Contains all the volume rendering functionality, included creating and deleting VBOs, calculating and rendering the 3D volume and exporting the volume to a file.

- *marchingCubes_kernel.cu:* Contains all the GPU computations, where there is mainly 3 parts of the file; the volume rendering part (the same as in [1]), the image processing part, and the threshold lookup part, which is used by both of the two other parts.

A.2 Main Functionality Flow

In this section, we will show the function's dataflow between the source files. Only the most important variables is described in the diagrams.

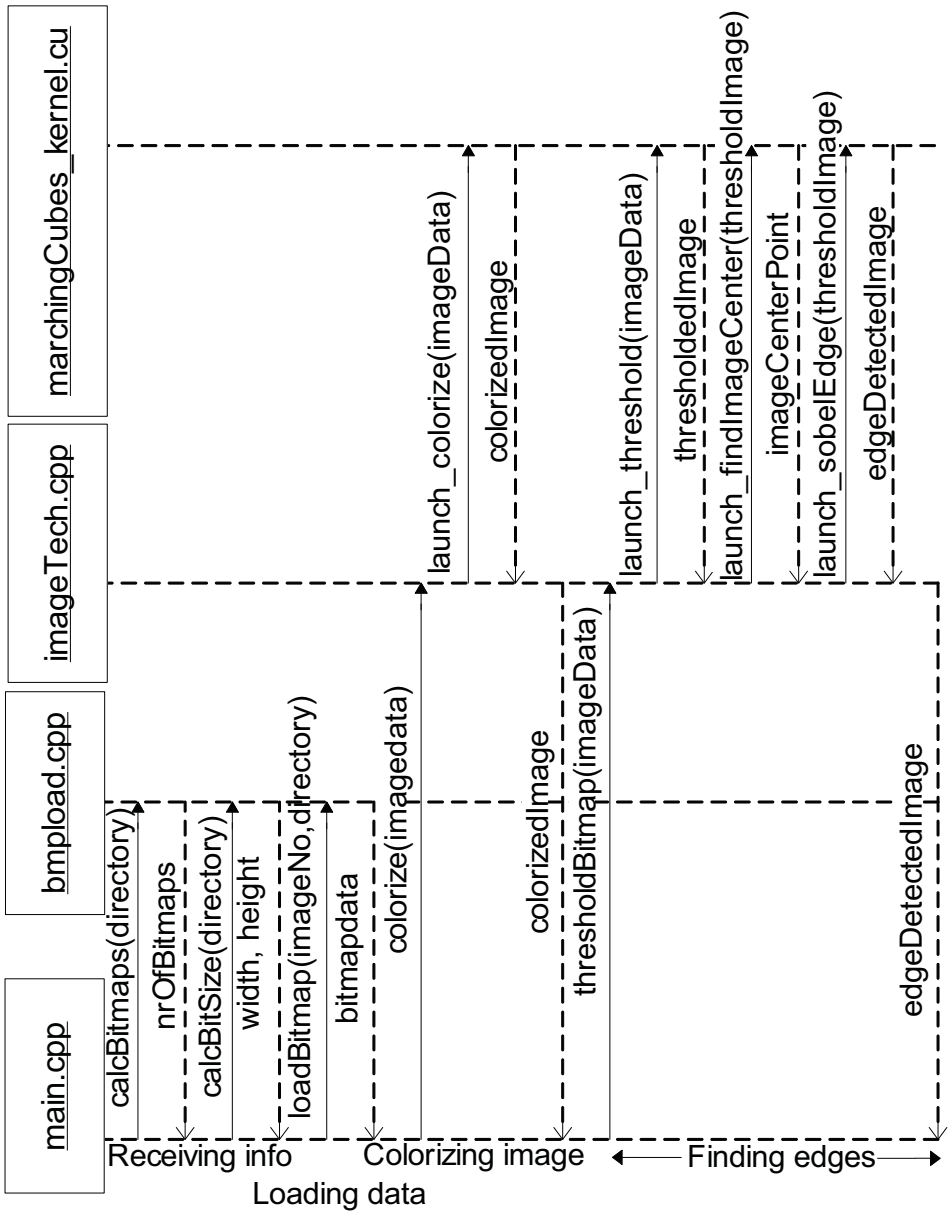


Figure A.1: Dataset loading flow sequence.

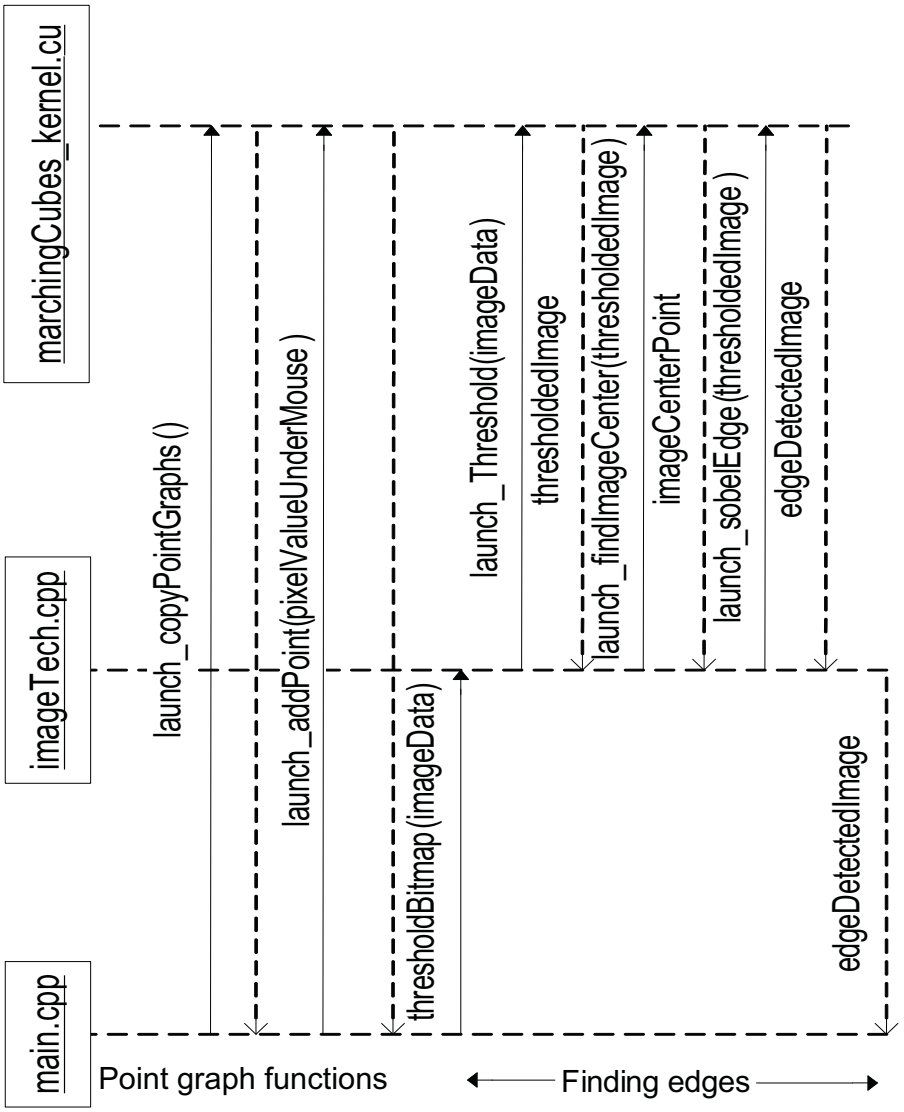


Figure A.2: Threshold change flow sequence.

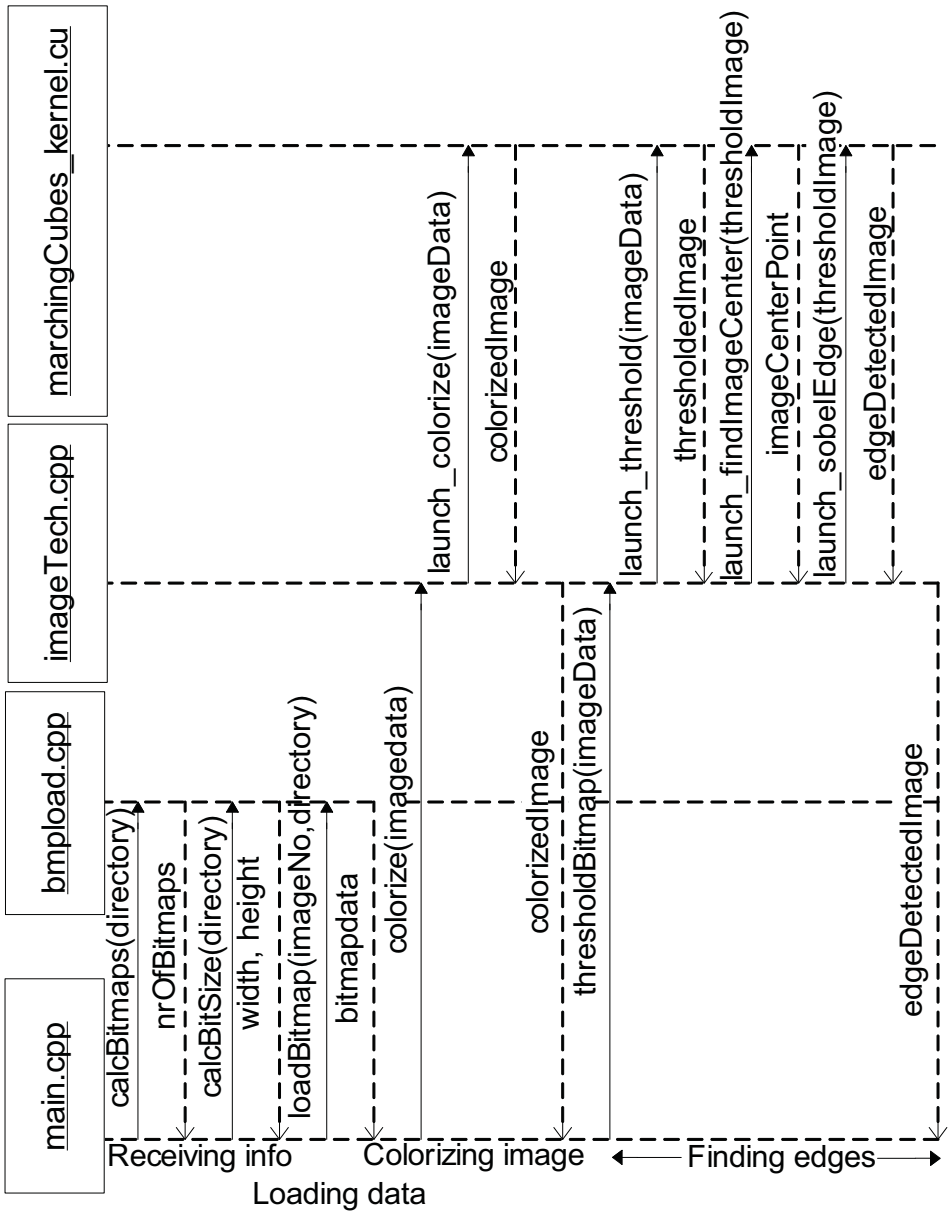


Figure A.3: Volume rendering flow sequence.

A.3 Further Details

For further details, we recommend to take a look at the source code, which is commented in functionality detail. More details is also explained in Chapter 5.

Appendix B

Software User's Guide

This appendix contains a short guide of how to use our program. Included in the guide is system requirements and a frame by frame program flow.

B.1 Hardware and Software Requirements

To run our program, the following hardware and software is needed:

- NVIDIA GeForce 8800 or newer GPU from NVIDIA with SM4 support
- 32bit Microsoft Windows OS (XP, Vista or 7)
- NVIDIA CUDA 2.1 Driver or newer
- (For using the exported data): Schlumberger Petrel 2008 or similar.

In addition for further development, there are additional software requirements:

- Microsoft Visual Studio 2005 or 2008
- CUDA SDK 2.1 or newer
- NVIDIA OpenGL SDK 10.5

B.2 User's Program Flow

In this section, we provide a screen flow of how to use the program. When the program starts up, the first screen loaded is the one in Figure B.1. For using any of the functionality implemented, a dataset needs to be loaded. In the further sections, we will explain the screen flow of each of the main functions.

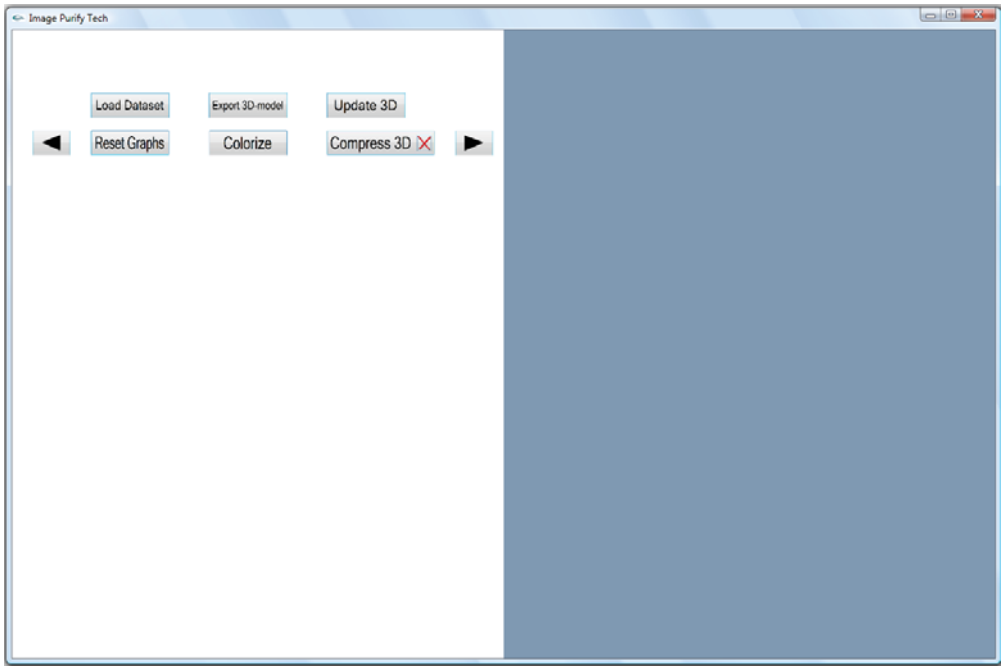
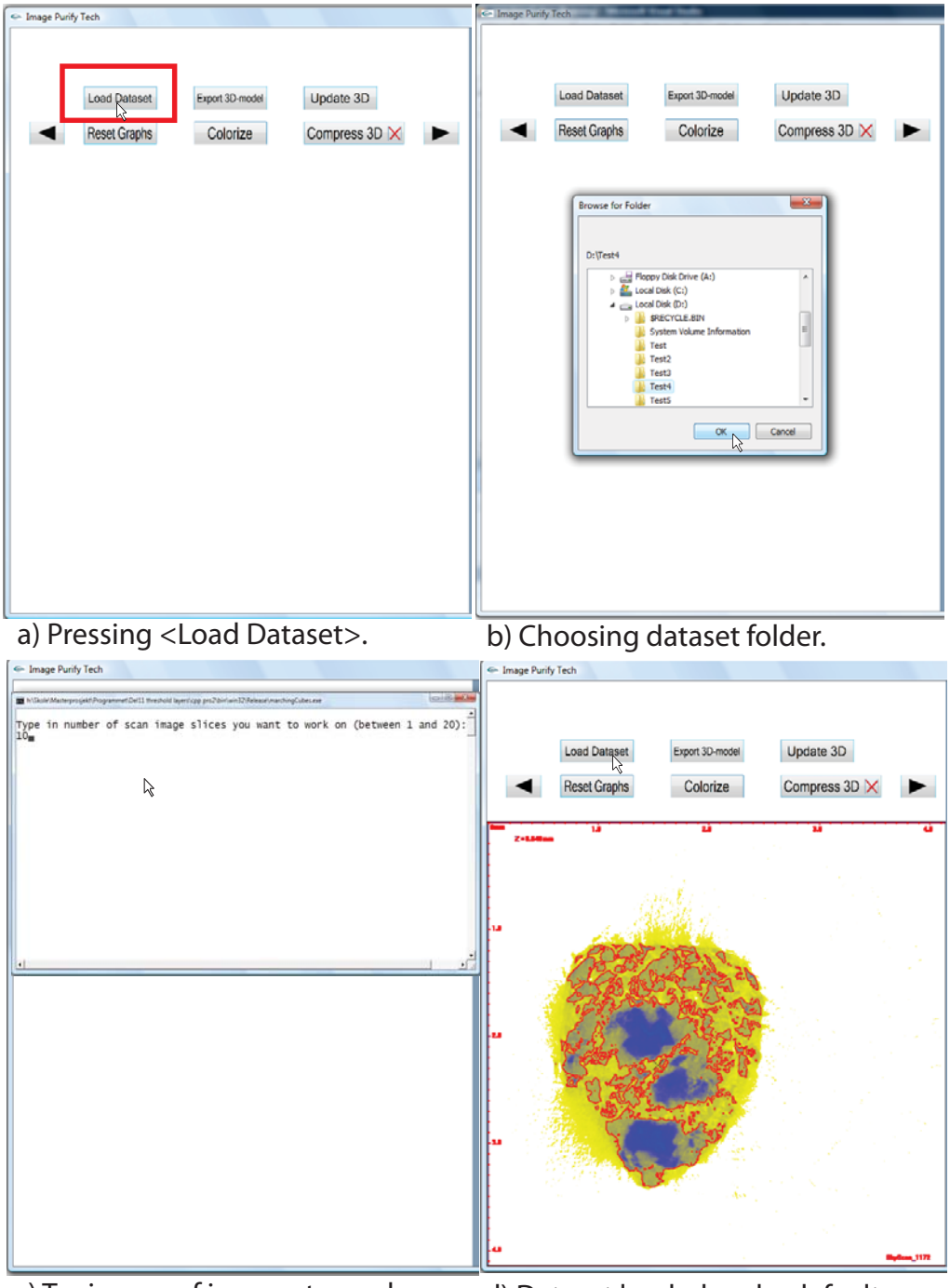


Figure B.1: Program startup screen.

B.2.1 Loading Dataset

To use any of the image processing or volume rendering techniques, a dataset needs to be opened. The first thing to do when starting a program, or starting working with a new dataset, is to press the button <Load Dataset>. The screen flow for loading a dataset is shown i Figure B.2. If the user should choose to load a folder without 8bit BMP images, the loading sequence will stop. When loading a dataset, the program automatically uses a global threshold, which will be reset after the first threshold adjustment.



a) Pressing <Load Dataset>.

b) Choosing dataset folder.

c) Typing no of images to work on, between 1-20, and press enter.

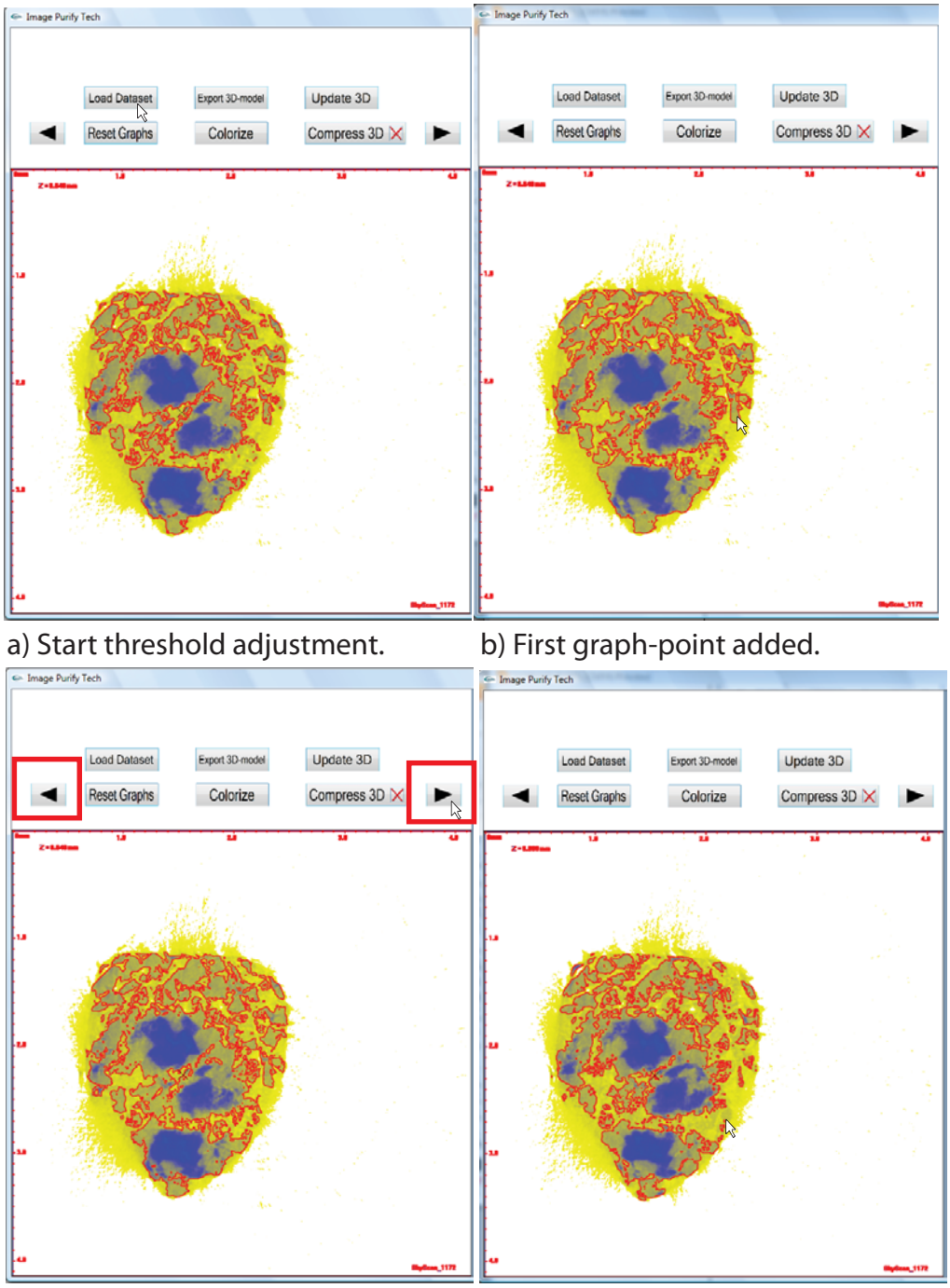
d) Dataset loaded and a default threshold used.

Figure B.2: Loading dataset - screen flow.

B.2.2 Adjusting Threshold

The dataset can be thresholded/segmented by using the mouse-pointer inside the 2D image loaded. By pressing the mouse-button, with the pointer inside the image, the thresholding-algorithms start running. When the user moves the mouse-pointer around the image, the thresholds will update with the image pixel under the mouse-pointer's density in a threshold-graph. The threshold calculations will happen per frame, while the mouse-pointer moves, until the mouse-button is unclicked. By unclicking the mouse button, the current threshold-state (graph) will be saved. These steps is illustrated in Figure B.3. By clicking the mouse button again, a second threshold reference point will be started, and added when the mouse button is unclicked again. When the user is satisfied with the current image, the next or previous image should be clicked (the buttons highlighted in Figure B.3c), and worked with the same way.

There are also some extra functionality for helping the user to make a good thresholding of the scan data images.



a) Start threshold adjustment.

b) First graph-point added.

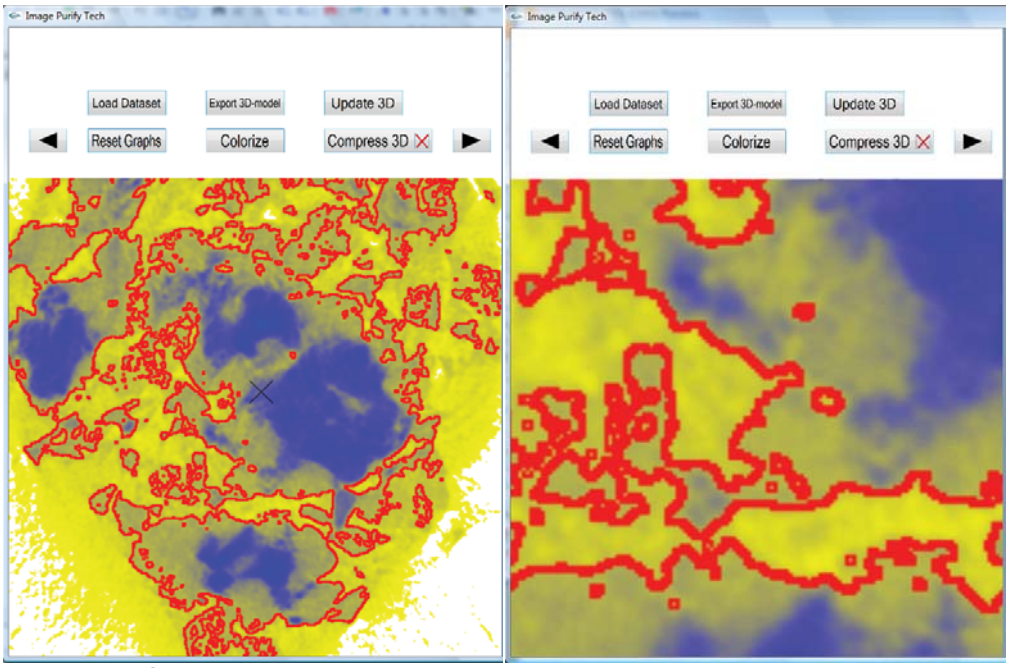
c) Thresholding complete. Clicking next image slice.

d) Next image slice loaded, and ready for thresholding.

Figure B.3: Threshold adjustment screen flow.

Zooming

In images with high resolution, there can be hard to identify the pore/rock edges. Therefore by using the mouse-wheel, the user is able to zoom in an out on the current image, like illustrated in Figure B.4. The user is also able to move over the image while zoomed in by pressing the mouse-wheel and moving the mouse while holding. While zoomed in, the threshold adjustment works the same way as before.

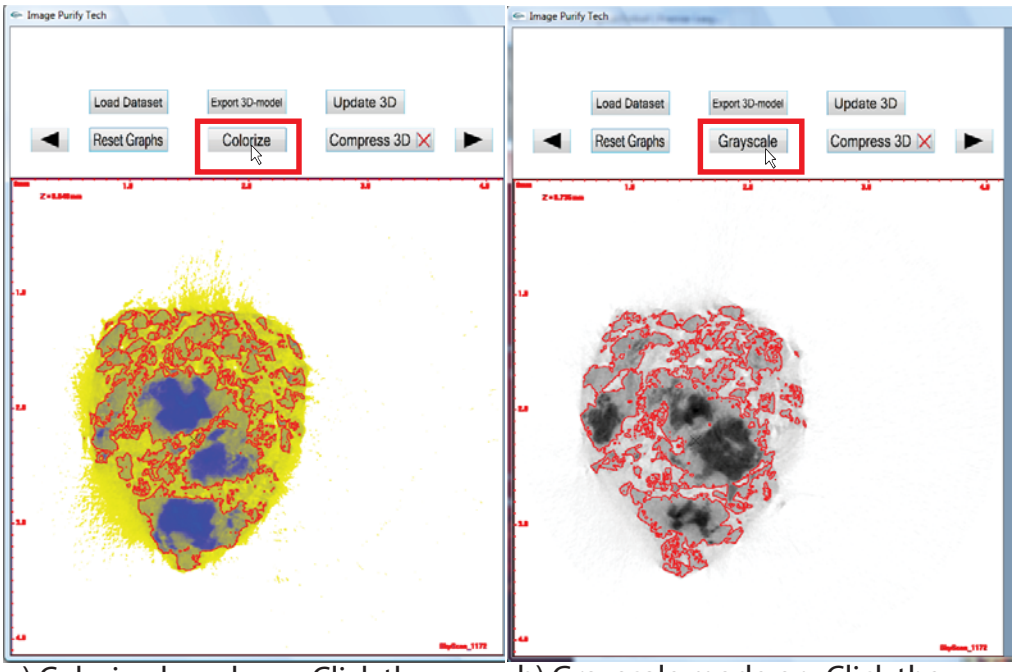


a) Zoomed in. b) Zoomed in even more.

Figure B.4: Threshold adjustment - zooming.

Density colors

Scans of different rock samples can often have a different average density. Therefore there also can be harder to observe the pores in the colorized image in some cases, than in the raw data. By pressing the button highlighted in Figure B.5, the user is able to change between grayscale and colorized mode of the images, without having any effect on the threshold-graphs.



a) Colorized mode on. Click the button changes mode.

b) Grayscale mode on. Click the button changes mode.

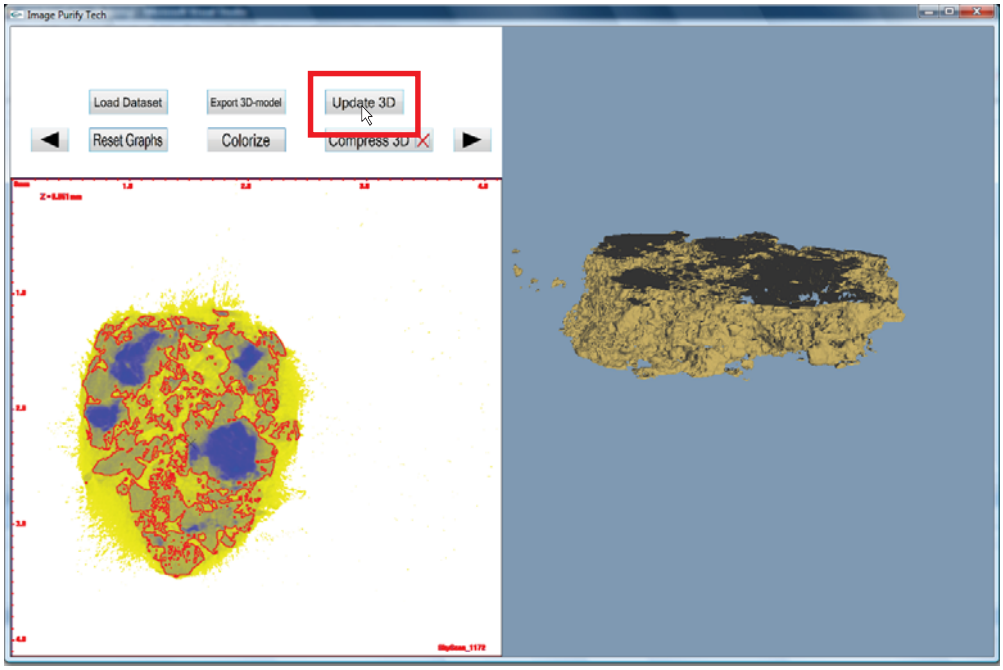
Figure B.5: Threshold adjustment - density colors.

Reset graphs

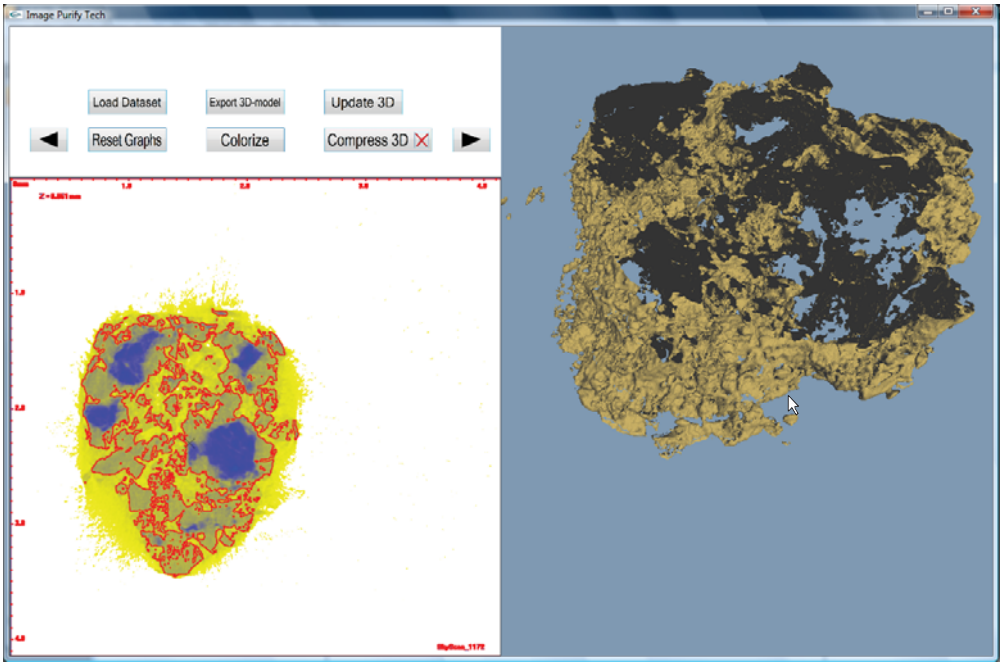
If the user is not satisfied with the thresholding-graphs he or she created, the <Reset Graphs> button can be pressed. The button will then reset the graphs for all the images.

B.2.3 Volume Rendering

When the user is satisfied with the threshold-graphs of all the images, the <Update 3D> button is the next to press. There will first be a little loading time, before the 3D model will show to the right of the screen. The user can also use the mouse to rotate the model. The steps for the volume rendering is illustrated in Figure B.6. The volume will be calculated for each time the user presses the <Update 3D> button, and therefore it is available to take a look at the model while working with it.



a) <Update 3D> pressed, and volume loaded.



b) Volume rotated.

Figure B.6: Volume rendering screen flow.

Compress

The 3D volume rendering requires a lot of GPU global memory. Therefore several GPUs can get a problem drawing the whole model, at least when using a large dataset. By pressing the <Compress 3D> button, the compress toggle turns on and off. Pressing <Update 3D> when the compress toggle is turned on, makes the dataset $\frac{1}{8}$th of the original size, and therefore is able to draw 8 times more of the current 3D model, as illustrated in Figure B.7.

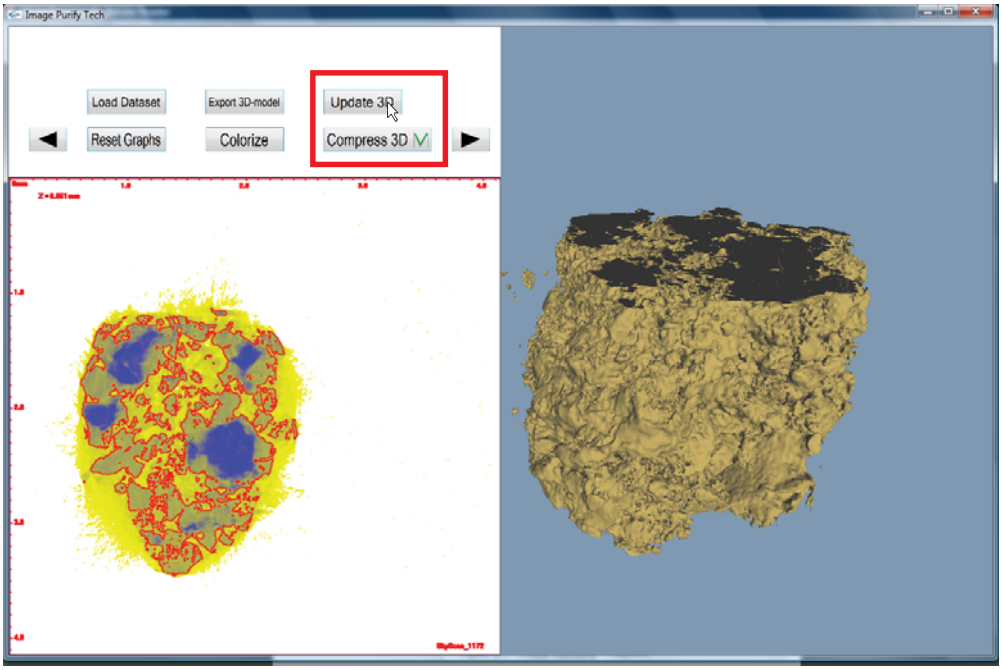
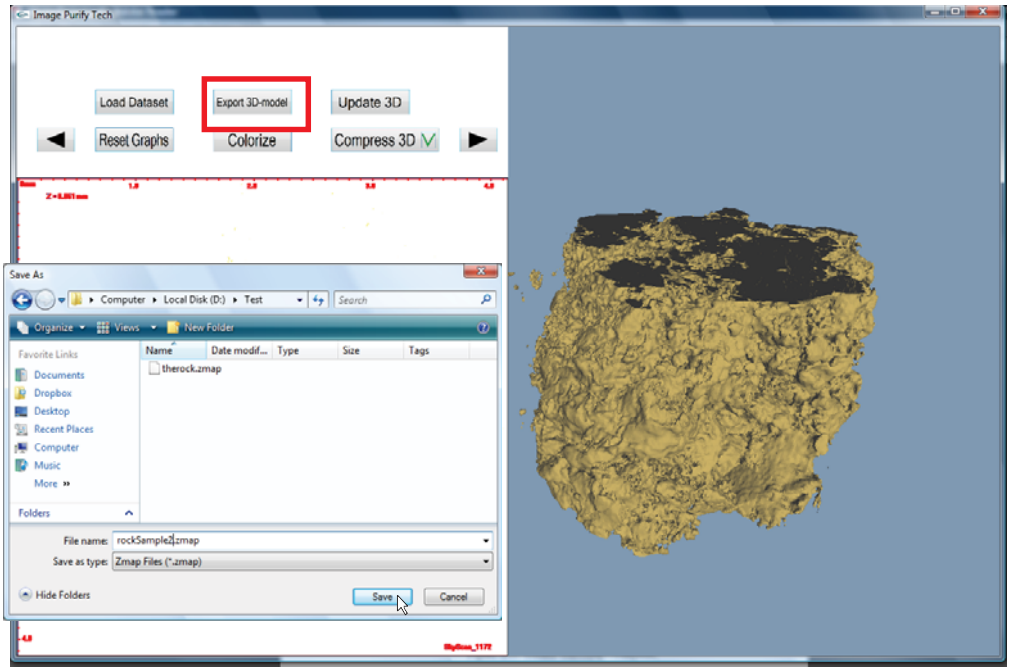


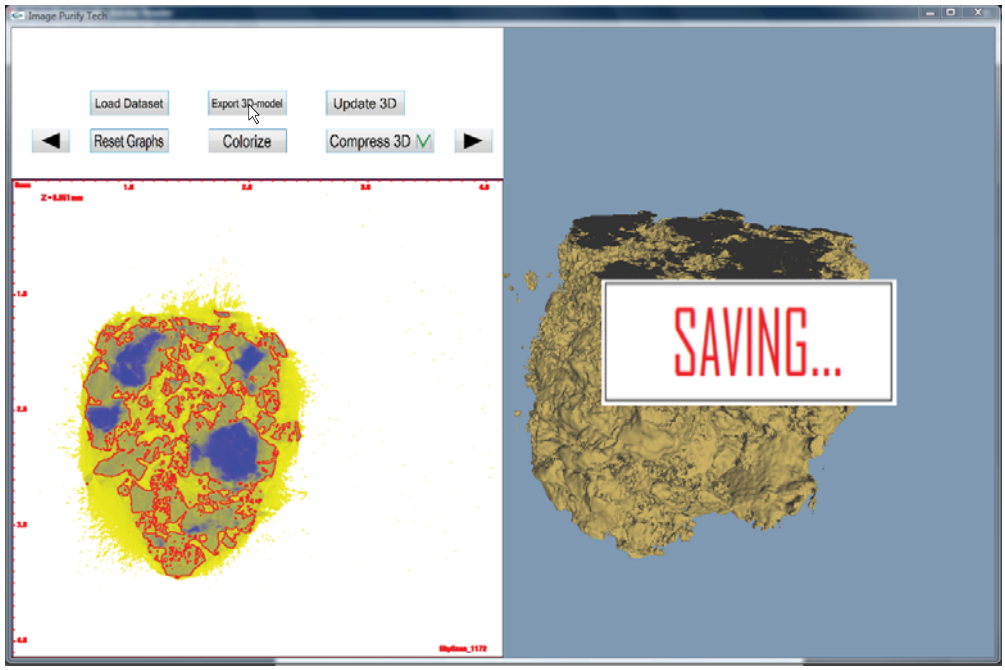
Figure B.7: Volume rendering - compress.

B.2.4 Export 3D

When the user is satisfied with the 3D model, the model can be exported to a file by pressing the <Export 3D-model> button. The user will then get a choice where to save, as illustrated in the screen flow in Figure B.8. The saving process uses several seconds, since all the data is first transferred from the GPU to the RAM, then further to the hard disk.



a) Export 3D-model pressed. Filename and location chosen.



b) Saving data will take several seconds.

Figure B.8: Export 3D screen flow.