Preface

This thesis is a documentation of our experimental work on using commodity graphics hardware for medical image segmentation. It is inspired by our pilot study, "*The GPU as a Computational Resource in Medical Image Processing*", carried out at NTNU fall 2004. The work has been carried out at the *Department of Computer and Information Science*, at the *Norwegian University of Science and Technology (NTNU)*, and in cooperation with *SINTEF Health Research*. The thesis is written by Harald Ueland and Martin Botnen[1], with professor Richard E. Blake at NTNU as teaching supervisor, and with guidance from Jon Harald Kaspersen at SINTEF.

We want to thank Richard E. Blake and Jon Harald Kaspersen for valuable advices and inspiration. We also give thanks to Arild Wollff at SINTEF for further motivation, and Trond Hagen, also at SINTEF, for giving us a framework that made programming GPUs more easily. This turned out to be very useful during our implementation. All data sets used in this work have been supplied by SINTEF.

The thesis is written in English using LaTeX. All code and programming functions mentioned in the text are presented in `TypeWriter` typeface.

Trondheim, June 2005

Harald Ueland   Martin Botnen

[1]{ueland — martibo}@idi.ntnu.no

Abstract

Modern graphics processing units (GPUs) have evolved into high-performance processors with fully programmable vertex and fragment stages. As their functionality and performance are still increasing, more programmers are appealed by their computational power. This has led to an extensive usage of the GPU as a computational resource in general-purpose computing, and not just within applications of the entertainment business and computer games.

Large volume data sets are involved when it comes to medical image segmentation. It is a time consuming task, but is important in the process of detection and identification of special structures and objects. In this thesis we investigate the possibility of using commodity graphics hardware for medical image segmentation. By using a high-level shading language, and utilizing state of the art technolgy like the framebuffer object (FBO) extension and a modern programmable GPU, we perform seeded region growing (SRG) on medical volume data. We also implement two pre-processing filters on the GPU; a median filter and a nonlinear anisotropic diffusion filter, along with a volume visualizer that renders volume data.

In our work, we managed to port the Seeded Region Growing (SRG) algorithm from the CPU programming model onto the GPU programming model. The GPU implementation was successful, but we did not, however, get the desired reduction in time consume. In comparison with an equivalent CPU implementation, we found that the GPU version is outperformed. This is most likely due to the overhead associated with the setup of shaders and render-targets (FBO) while running the SRG. The algorithm has low computational costs, and if a more complex and sophisticated method is implemented on the GPU, the computational capacity and the parallelism of the GPU may be more utilized. Hence, a speed-up in computational time is then more likely to occur compared to a CPU implementation. Our work involving a 3D nonlinear anisotropic diffusion filter strongly suggests this.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 2

2 Background and Related Work **3**

2.1 The Graphics Processing Unit (GPU) 3

2.1.1 The Graphics Pipeline 4

2.1.2 The GPU Programming Model 4

2.1.3 Hardware Constraints 7

2.2 High-Level Shading Languages 10

2.2.1 The Cg Language 10

2.3 Medical Image Segmentation 13

2.3.1 Seeded Region Growing 15

2.4 Related Work 16

2.4.1 Filtering on the GPU 17

2.4.2 Image Segmentation on the GPU 19

2.5 CustusX 21

3 Design **23**

3.1 Hardware Platform 23

3.2 Software 23

3.2.1 Graphics Library 23

3.2.2 A General GPU Framework 24

3.2.3 Framebuffer Object Extension 24

3.2.4 Image Format and Textures 25

3.2.5 Alpha-Testing 27

3.3 Filtering on the GPU 27

3.3.1 Median Filter 28

3.3.2 Anisotropic Diffusion Filter 28
3.4 Seeded Region Growing on the GPU 31
3.4.1 Exclude 32
3.4.2 Set Seed 32
3.4.3 Grow Region 32
3.4.4 Termination 33
3.4.5 Shade Segmented 33

4 Implementation **35**
4.1 Excluding Fragments 37
4.2 Seed Determination 39
4.3 Grow Region 40
4.4 Shade Segmented 40
4.5 Computation Mask 41
4.6 Termination 41
4.7 Filters 42
4.7.1 Median Filter 43
4.7.2 Nonlinear Anisotropic Diffusion Filter 44

5 Visualization **47**
5.1 Volume Rendering 47
5.2 Rotation 49
5.3 Examples 53

6 Results **55**
6.1 Test Setup 55
6.1.1 Hardware Platform 55
6.1.2 Graphical User Interface 55
6.1.3 Test Data Sets 56
6.1.4 Relevant Structures 56
6.1.5 Achievement and Evaluation of the Results 57
6.2 Results 58
6.2.1 Aorta 58
6.2.2 Brain Tumour 59
6.2.3 Cerebral Cortex 61
6.2.4 Liver 61
6.2.5 Kidneys 62

6.3 Comparison with CPU Implementation 63
6.4 Result Visualizations 64

7 Conclusions 73
7.1 Discussion 73
7.2 Final Conclusions 75

8 Future Work 77
8.1 Further Implementations 77
8.2 Integration into CustusX 78

Bibliography 79

A Seeded Region Growing - Cg Shaders 83
A.1 Exclude 83
A.2 Set Seed 84
A.3 Grow Region 85
A.3.1 Grow Region 3D 85
A.3.2 Grow Region 3D Using a Delta Value 86
A.3.3 Grow Region 2D 87
A.4 Shade Segmented 88

B Filtering - Cg Shaders 89
B.1 Median filter 89
B.2 Anisotropic Diffusion Filter 91
B.2.1 2D Implmentation 91
B.2.2 3D Implementation 93
B.3 Threshold 95

List of Figures

2.1 The graphics hardware pipeline architecture. 4
2.2 The graphics hardware pipeline based on the OpenGL API. 5
2.3 The GPU programming model 6
2.4 Programmable graphics pipeline 6
2.5 The GPU memory layout 8
2.6 The CPU and GPU analogy 9
2.7 The scatter and gather property 9
2.8 The execution of a GPU program 10
2.9 Simple threshold Cg program 11
2.10 Assembly code for the threshold shader 11
2.11 Segmentation with snakes 15
2.12 A segmentation of the cerebral cortex 21
2.13 Custus X 22

3.1 Framebuffer Object architecture 26
3.2 OpenGL fragment tests 27
3.3 The approximate version of median filtering. 28
3.4 Overview of our seeded region growing algorithm. 31

4.1 UML class overview 36
4.2 Optimization in the **Exclude**-shader. 39
4.3 Optimization in the Median-shaders. 43
4.4 2D Sobel Operators. 44
4.5 3D Gradient Operator. 45

5.1 Stages in the volume visualization 48
5.2 Rendering and blending a volume 50
5.3 Rendered volumes 53

6.1 Screen shot of the GUI 56
6.2 Measuring total time spent in hardware. 58
6.3 Image slice from data set A1 60
6.4 Result of using the anisotropic diffusion filter. 60
6.5 Segmentation of aorta in data set A1. 64
6.6 Segmentation of aorta in data set A2. 65
6.7 Segmentation of aorta in data set A3. 66
6.8 Segmentation of aorta in data set A4. 67
6.9 Segmentation of brain tumour in data set H1. 68
6.10 Segmentation of cerebral cortex in data set H1. 69
6.11 Segmentation of liver in data set A2. 70
6.12 Segmentation of kidneys in data set A4. 71

List of Tables

2.1 Relevant standard library functions in Cg. 13

6.1 The data test sets that we use. 57
6.2 Segmentation settings and achieved computation times. 59

Chapter 1

Introduction

In this chapter we present reasons for using the GPU for general purpose computing, and specifically why medical image segmentation can benefit from using graphics hardware.

1.1 Motivation

The main application of graphics processing units (GPUs) has been in entertainment business and computer games, enabling fast rendering of anti-aliased, textured and shaded geometric primitives. As their performance and functionality have been increasing and now give support for floating-point computations, and the existence of compilers for high-level programming languages, many new algorithms and applications have been suggested. These try to take advantage of the parallelism and vector processing capabilities of the GPUs.

The reasons for focusing on GPU implementations are numerous. One reason is that, if you are already planning to visualize your data, you can remove unecessary data stream from the central processing unit (CPU) to the GPU. The GPU offers a parallel model for programming with internal parallel operations which practically is a broad vector model with varying vector and matrix sizes, and the implementations are often small and surveyable. The GPU also has a bigger slope with respect to performance, measured in GFLOPS, than the CPU, and it gives access to high internal bandwidth. When programming, we only have to focus on exploiting this internal bandwidth. If more bandwidth is needed, a very big cluster of CPUs would be necessary. At last, GPUs offer the possibility of transparent compression and efficient improvement in the later generations of hardware.

Using the GPU as a co-processor can in light of this improve many medical image processing tasks. The CPU is relieved and can be used for other tasks as the GPU is left with much of the computational work. Medical image segmentation is a very time consuming task where an enormous amount of data must be processed. As the segmentation often serves as a basis for further computations and examinations, a speed-up in this step will have a large impact on the overall process of many medical image processing tasks. Hopefully, a possible, small loss in accuracy by using the GPU instead of the CPU is acceptable, and many medical tasks may find an advantage in segmenting data quicker.

Seeded region growing (SRG) is a simple, surveyable algorithm that we believe can be mapped onto the GPU. Of course, the preferred segmentation method varies with the actual problem, but for many cases SRG can produce acceptable results with its low computational costs.

1.2 Thesis Outline

In Chapter 2 we start with some general theory about the GPU, before we move on to medical image segmentation and look at some existing, related work that are relevant for our task. This is followed by Chapter 3, where we deal with the design of our segmentation application. Here we mention the hardware platform that we operate within, and present the technology and software methods that will be used in our implementation. The chapter is completed with an overview of how the different filtering techniques and the seeded region growing algorithm can be mapped onto the GPU. In Chapters 4 and 5, we give a detailed description of our implementation of SRG and how the data are visualized, respectively. Then we present our results in Chapter 6 prior to discussions and conclusions in Chapter 7, and remaining, future work in Chapter 8.

Chapter 2

Background and Related Work

2.1 The Graphics Processing Unit (GPU)

Traditionally, graphics processing units, GPUs for short, were designed and used almost exclusively to perform graphics-oriented tasks. From being a simple memory device, a modern GPU is now a fully programmable parallel processor. It consists of two programmable hardware units and a number of units that are not directly accessible to the application developer. Modern graphics cards have an enormous amount of processing power which could be harnessed for doing different calculations and visualizations. The computation power of graphics hardware has increased at a higher rate than that of general purpose hardware, and its cost has dropped so much that it has become available in every PCs on the marked. Today, commodity graphics hardware is used in production of video games and computer generated movies, computer aided design and scientific visualization, and even to solve non-graphics-related problems. By exploiting the GPU, some problems may be accelerated by over an order of magnitude over the CPU. For example, a 3 GHz Pentium 4 CPU has a theoretical performance of 6 GFLOPS whilst 40 GFLOPS has been observed for the GeForce 6800 Ultra from NVIDIA[1] [16].

Throughout this work we will use the terms *pixels, voxels, texels* and *fragments*. A *pixel* is an atomic part of a 2D image whilst a *voxel* is the corresponding part of a 3D volume. A *texel* is a part of a texture and will be equivalent to pixels in the design. A *fragment* is what the GPU fragment processor operates on, and will be used interchangeably with pixel when this is appropriate in the context.

[1]Observed on a synthetic benchmark.

2.1.1 The Graphics Pipeline

The GPU uses a pipeline architecture to process multiple fragments in parallel. Several units are working in parallel on different vertices and fragments at different stages of their transformations into pixels. GPUs are optimized for 4-component vector operations, with both MIMD[2] (vertex) and SIMD[3] (pixel) pipelines. Figure 2.1 shows the graphics hardware pipeline architecture, split into three functional stages.

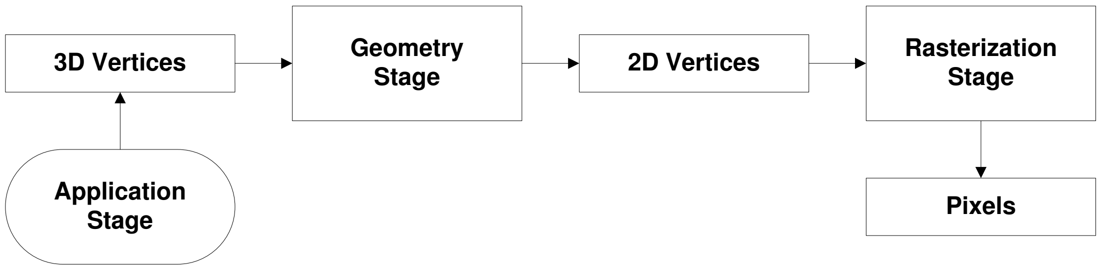


Figure 2.1: The graphics hardware pipeline architecture.

The application stage outputs 3D vertices. Every vertex has some attributes attached to it. This include 3D position, orientation, texture coordinates and colour. Mathematical operations are calculated on those values at the different stages in the pipeline. In the geometry stage, projection transforms transform these vertices into 2D primitives. Finally, the rasterization process computes pixel values to form the final image by interpolating the attributes of the incoming vertices. Texels can be computed from pre-stored textures and texture coordinate attributes. They can be combined with the colour attributes of the fragments for any computation purposes. The resulting pixel value is written into the framebuffer either by replacing or blending with the old value.

2.1.2 The GPU Programming Model

As of today, there are two major 3D APIs. OpenGL, which we are using in our work, is maintained by the OpenGL Architectural Review Board (ARB[4]) composed of several companies. The second API, DirectX, is maintained by Microsoft

[2]Multiple Instructions/Multiple Data.
[3]Single Instruction/Multiple Data.
[4]See http://www.opengl.org/about/arb.

Corporation, but is compatible with the Windows operating system only. DirectX is very popular in the gaming industry. Figure 2.2 shows the graphics pipeline based on the OpenGL API.

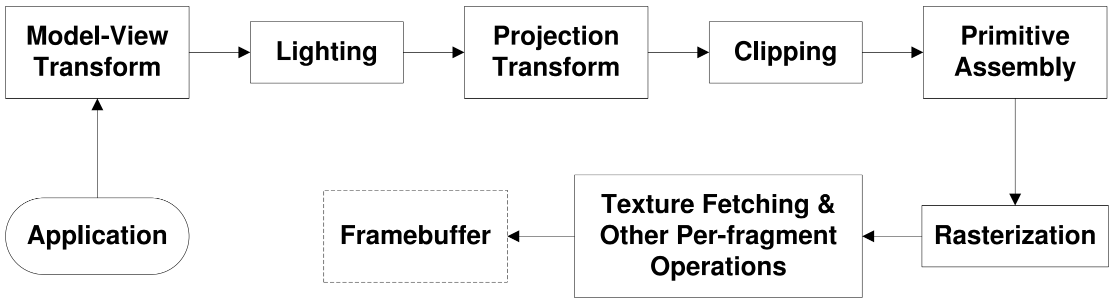


Figure 2.2: The graphics hardware pipeline based on the OpenGL API.

As mentioned, the GPU consists of two programmable hardware units. This means that stages in the graphics pipeline are made explicit to the application programmer, and developers can write their own shaders where they specify the operations they want to be executed on a per-vertex and -fragment basis during the rendering. Such vertex- and fragment programs, also called shaders, can be written in high-level programming languages that most modern GPUs support. Examples of such high-level languages are given in Section 2.2.

The programmable processors in the processing pipeline are the *vertex processor* and the *fragment processor*. Figure 2.3 shows the GPU programming model. The programmable parts are shaded.

When you make a CPU application that utilizes graphics hardware, you will normally issue calls to the hardware via a 3D API. The API will translate these calls into a stream of GPU commands and data, all of which are sent to the GPU front end across the CPU/GPU boundary. Once the transfer has been completed, CPU and GPU execution can continue asynchronously.

The programmable *vertex processor* is a unit that operates on incoming vertices. It performs traditional vertex and normal transformations, texture coordinate generation and transformation, lighting and colour material applications. Programs that are intended to run on this processor are called *vertex shaders* or *vertex programs*. Vertex shaders can be used to specify a completely general sequence of operations to be applied to each vertex. On the other hand, they can not perform graphics operations that require knowledge of other vertices at a time or that require topological knowledge, e.g. perspective division, backface culling

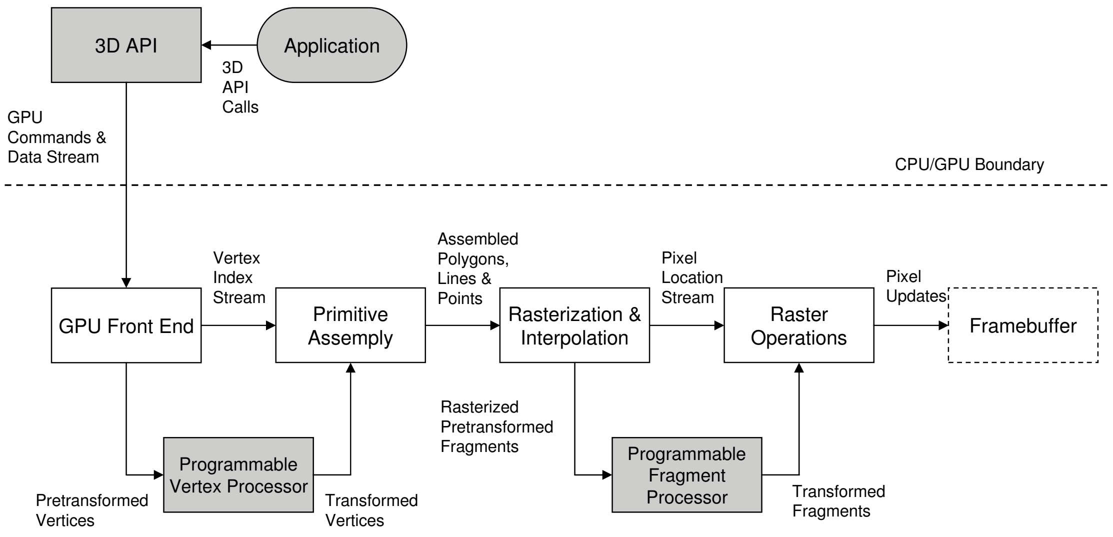


Figure 2.3: The GPU programming model [6].

and depth range.

The processor operates on one vertex at a time. Its design is focused on the functionality needed to transform and light a single vertex. The output of the vertex processor is sent through subsequent stages of processing before the *fragment processor* performs its operations. This is shown in Figure 2.4.

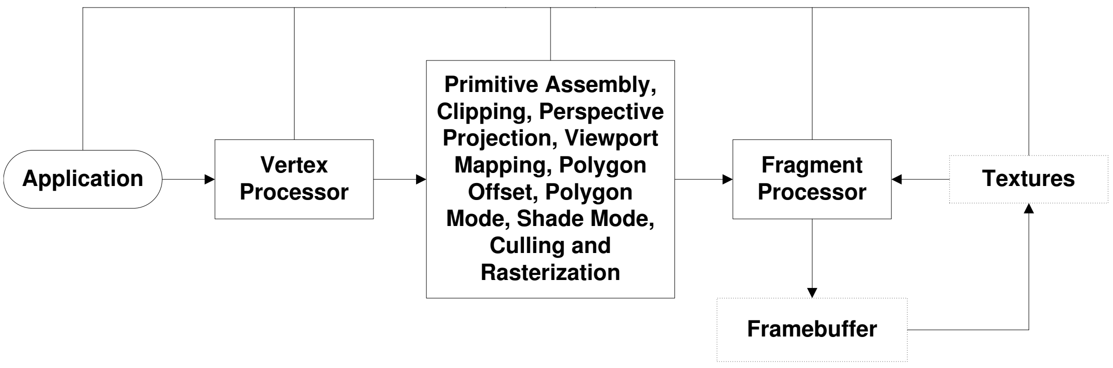

Application
Vertex Processor
Primitive Assembly, Clipping, Perspective Projection, Viewport Mapping, Polygon Offset, Polygon Mode, Shade Mode, Culling and Rasterization
Fragment Processor
Textures
Framebuffer

Figure 2.4: Programmable graphics pipeline. The application has control of both the vertex- and fragment processor, indicated by the lines connecting them. Arrows represent data streams.

The *fragment processor* operates on fragments values and their associated data. The fragment processor is intended to perform operations on inputs from the rasterization stage. This include operations on interpolated values, texture access and texture application, fog, colour sum and point size. Programs that run on this processor are called *fragment shaders* or *fragment programs*. Fragment shaders can be used to specify a completely general sequence of per-fragment operations to be applied to each fragment passing through the processor. Every fragment is invisible to all the others, so the fragment shaders can not perform graphics operations that require knowledge of other fragments. This programmable unit can only write to the framebuffer. It does not have read capability. However, it does have the capability of texture lookup.

We have implemented our segmentation algorithm on the NVIDIA GeForce 6200 GPU with 128 MB of video memory. The GeForce 6 series shader architecture consists of two shading units per pixel. This superscalar architecture delivers a twofold increase in pixel operations in any given clock cycle. With 16 pixel pipelines and 8 pixel shader operations per pixel, this means 128 pixel shader operations per clock cycle [7]. This shows that a GPU based solution is likely so accelerate computations when they can be done on independent elements, i.e. the computations can be mapped onto a streaming model.

2.1.3 Hardware Constraints

In order to produce efficient and running GPU code, there are a number of hardware induced properties and constraints that need to be considered. The GPU memory layout and how this memory is accessed is crucial when writing programs that are intended to run on graphics hardware. It is also important to know where performance bottlenecks in a graphics application may be located. They may reside on the CPU or the GPU. An overview of the current GPU memory layout is shown in Figure 2.5.

The GPU memory is divided into *Video Memory* and *On-Chip Cache Memory*, a design that very much impacts the way in which texture memory can be efficiently accessed.

When a shader is executed, it operates on one vertex or fragment at a time. The fragment processor can read from anywhere in the input textures but it can only output on the very pixel it operates on. The processor is said to be able to *gather* but not *scatter*, as depicted in Figure 2.7. The vertex processor on the other hand, can change vertex positions, texture coordinates, and a few other values. The newest graphics cards have the ability to also read from texture memory from the vertex processor.

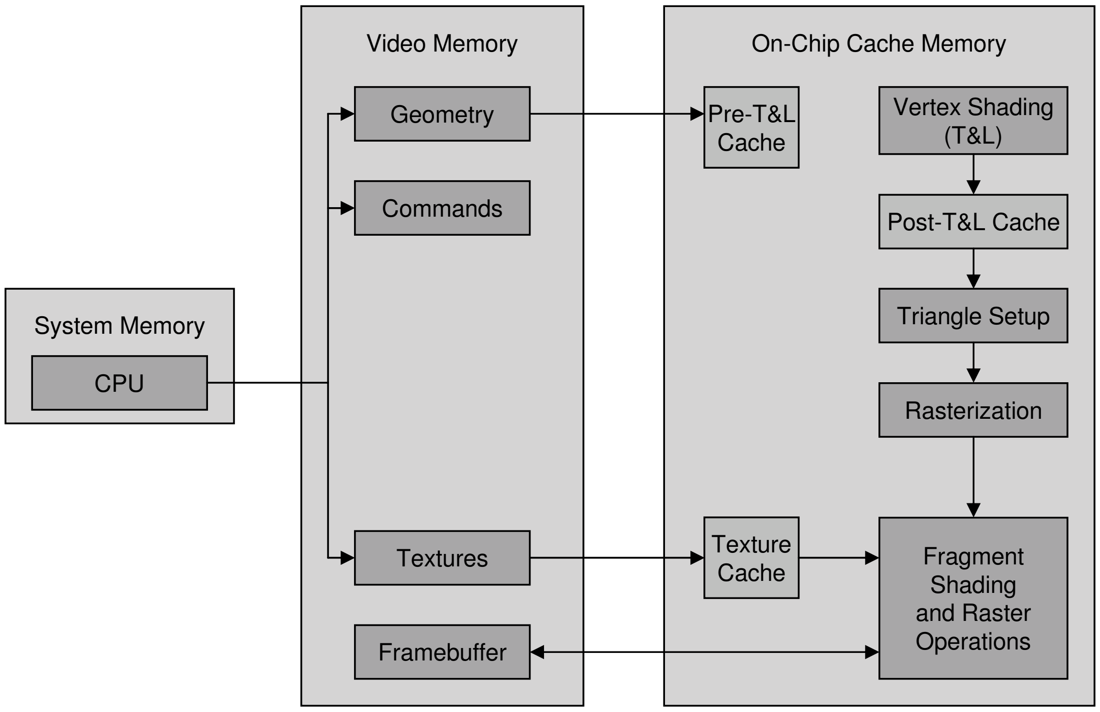


Figure 2.5: The GPU memory layout [8].

To see the differences between CPU- and GPU programming we also have to consider the GPU state access. There are no global registers incorporated in any current hardware. All registers are temporary and reset before each record is processed. This stateless property makes it difficult to implement even simple algorithms that need to store values after each calculation, like counting or averaging. To overcome this, calculated values after each rendering pass have to be output to the render-target and read back to texture memory before the next pass. Using the *render-to-texture* extension (See Section 3.2.3), the output from the fragment processor is automatically rendered to a bound texture. This texture is then input in the next pass. The CPU and GPU analogy is depicted in Figure 2.6.

When it comes to data transfer between the CPU and GPU, current hardware uses AGP[5] or PCI Express technology to communicate between main memory and video memory. This communication is expensive and therefore it is important to keep as much computation on the GPU for as long as possible.

[5]Accelerated Graphics Port

```
// A CPU implementation
int img[IMG_SIZE];

// Impossible in the same GPU program.
for (int i=0; i<IMG_SIZE; i++) {
    img[i] = Do some calculations;
    ...
}

// The GPU analogy
for all pixels in image
do
    // Fragment program
    Perform per-pixel operations
done
```

Figure 2.6: The CPU and GPU analogy. On the GPU, there is a lack of the random access write capability of high-level arrays. A fragment program maps to loop internals over all pixels.

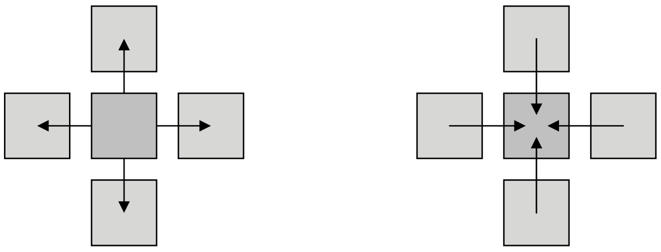


Figure 2.7: Left: Scatter write to different parts of the render *target*. Right: Gather read from different parts of render *input*.

2.2 High-Level Shading Languages

Programs on GPUs have traditionally been written in assembly code, which denotes a lot of work. A high-level programming language raises the level of abstraction so that low-level issues, e.g. register allocation, is at no concern for the programmer. It also gives the programmer the ability to develop GPU programs with familiar constructs and syntax with no thought to hardware details. Several compilers of high-level shading languages have been released, such as OpenGL Shading Language (glslang), BrookGPU and the high-level metaprogramming language Sh [22, 4, 3, 9]. Cg is also a high-level shading language developed by NVIDIA corporation [6].

Generally, a program written for the GPU is compiled to GPU assembly code, loaded and executed as listed in Figure 2.8.

```
Compile program
Download program to the GPU
Enable shader/processor
Render geometry
Disable shader/processor
Display result
```

Figure 2.8: The compiling and execution of a GPU program.

Once a program is bound, it will execute in all subsequent drawing calls for every vertex (for vertex programs) and fragment (for fragment programs).

2.2.1 The Cg Language

Cg, *C for Graphics*, offers the ability to write programs that can be compiled into optimized GPU code, either in advance or on demand at run time. Cg is based on C, and much of the syntax for declarations, function calls and datatypes are similar. As GPUs normally have at least two programmable processors, and CPUs normally have one processor, there are of course some differences. However, the basic idea is the same: "*Just as C was derived to expose the specific capabilities of processors while allowing higher-level abstraction, Cg allows the same abstraction for GPUs.*" [6]. This makes it possible for developers to focus on the ideas, concepts and the effects which they want to create with less attention to the hardware implementation. Cg programs are portable by the fact that they can run on any operating system, platform and graphics hardware. The reason for that is among other things that it is a functional language rather than hardware implementation specific.

Figure 2.9 and Figure 2.10 show a simple Cg shader that does basic thresholding and the corresponding GPU assembly code.

```
void threshold(
    float2 texCoord : TEXCOORD0,
    uniform float threshold,
    out float4 color : COLOR,
    uniform samplerRECT texture)
{
    color = texRECT(texture, texCoord);
    color.rgb = color.rgb * step(threshold, color.r);
}
```

Figure 2.9: Cg shader performing simple thresholding.

```
DECLARE threshold;
TEX   R0, f[TEX0], TEX0, RECT;
SGER  H0.x, R0, threshold;
MULR  o[COLR].xyz, R0, H0.x;
MOVR  o[COLR].w, R0;
END
```

Figure 2.10: Assembly code for Cg shader shown in Figure 2.9.

Additional Advantages

Compared to the other high-level shading languages, Cg has some advantages. First of all, it gives support for both of the hardware APIs, DirectX and OpenGL. Cg also uses *binding semantics*, i.e. pre-defined structures, that are used when sending data packets between the application and a vertex program, and for vertex shader to fragment shader communication. The binding semantics represent an easy accessible way of communicating between the different processing units.

Vertex attributes such as colour, normal vector, and texture coordinates are examples of what is contained in the binding sematics between the API and the vertex shader. These attributes can then be transformed, and the output from the vertex shader is stored in the binding semantics between vertex shader and fragment shader. The fragment shader gets colour values and texture coordinates as varying input, and colour and depth values are its output.

Language Profiles

All CPUs support essentially the same set of basic capabilities, and a common programming language like C supports this set on all CPUs. However, GPU programmability has not quite yet reached this same level of generality. Different graphics hardware and the vertex- and fragment processors support different capabilities. Cg addresses this by introducing the concept of language *profiles*. A Cg profile defines a subset of the full Cg language that is supported on a particular hardware platform or API. These optional language features include certain control constructs and standard library functions. The language profiles also defines the precision of the `float`, `half` and `fixed` datatypes. Examples are the DirectX 9 vertex- and fragment profiles and the OpenGL ARB vertex- and fragment profiles.

Useful Standard Library Functions and Special Operators

In addition to the binding semantics, the Cg language has some built-in functions like geometric, mathematical, and texture map functions that simplify the GPU programming even more. These are for the most optimized for programming the GPU and are executed very quickly, hence it is efficient to use them whenever possible. Some examples of the standard library functions that are relevant for our implementation are given in Table 2.1.

Conditionals are not effectively implemented on the GPU as it tends to run at the same speed as if both the `if` and the `else` part were executed, so the crux for faster Cg code is to avoid them whenever possible. If conditionals can not be avoided, the most effective condition is the `step`-function. This function can be used if a result depends on a comparison of two values, and is efficiently implemented in hardware.

The `texRECT`-function is a standard non-projective texture lookup, where `samplerRECT` is the *texture sampler*, and the `float2` denotes a two-vector texture coordinate.

Cg has a special operator called the *swizzle* operator, which is denoted by a dot

Function	Description
`abs(x)`	The absolute value of `x`.
`all(x)`	Returns `true` if every component of `x` is not equal to 0. Returns `false` otherwise.
`any(x)`	Returns `true` if any component of `x` is not equal to 0. Returns `false` otherwise.
`exp(x)`	Exponential function, e^x.
`max(a, b)`	Maximum of a and b.
`min(a, b)`	Minimum of a and b.
`step(a, x)`	0 if `x` $< a$, 1 if `x` $>= a$.
`texRECT(samplerRECT tex, float2 s)`	2D `RECT` nonprojective. Returns a float4 value.

Table 2.1: Relevant standard library functions in Cg.

(.). It makes it possible to rearrange, repeat, and omit the components within a vector to form a new vector. The elements are addressed by the characters x, y, z and w, respectively, but r, g, b and a can be used in the same manner. For example, `float4(a, b, c, d).yxzz` yields the vector `(b, a, c, c)`. You can create a vector of a scalar in a similar way, where `a.xxxx` yields the vector `float4(a, a, a, a)`. This operator is implemented efficiently in the GPU hardware with no performance penalty.

When the swizzle operator is used on the left hand side of an assignment statement, it serves as a write mask operator, e.g. `color.ra = float2(1.0f, 0.5f);` sets the red component to 1.0 and the alpha value to 0.5 and leaves the other components unchanged.

Using vectors instead of scalars is highly appreciated when programming GPUs, as they can perform four arithmetic operations as quickly as one single operation. Hence, a more vectorized code is preferred, although the compiler tries to vectorize the source code for you.

2.3 Medical Image Segmentation

Image segmentation is a difficult task, and in order to achieve good results, several issues must be dealt with. Hence, much effort in research has addressed these problems, and this has led to an emergence of many segmentation techniques and algorithms. Despite this, image segmentation remains as a grand challenge in computer vision, and no segmentation technique is superior to other. Thus, the choice of technique must be based on the actual problem.

The main goal of image segmentation is to subdivide the image into its disjoint regions or objects. In medical image segmentation, the algorithms operate on 2D intensity images, or 3D data, from medical scanners, e.g. X-ray, Magnetic Resonance (MRI) and Computed Tomography (CT) scanners. We can distinguish between four popular segmentation techniques that are used on such intensity images.

- **Threshold techniques** are based on local pixel information and are useful when the intensity levels of the objects fall squarely outside the range of levels in the background.
- **Edge-based methods** try to detect contours in an image. Blurring may introduce some difficulties in connecting broken contour lines.
- **Region-based methods.** Here, neighbouring pixels of similar intensity levels are usually grouped to form connected regions in the image. This is the image *partitioning* part. Adjacent regions are then merged under some criterion involving perhaps homogeneity or sharpness of region boundaries. This method may cause fragmentation if the merging criteria is too stringent, and may overlook blurred boundaries and overmerge if the criteria is too lenient.
- **Connectivity-preserving relaxation-based segmentation methods,** usually referred to as *active countour models*, use the concept of energy minimization. An initial boundary shape represented in the form of spline curves is guided by various shrink and expansion operations according to some energy function. Snakes are active contour models. Areas of application include detection of edges, lines, and subjective contours, motion tracking and stereo matching [21]. Figure 2.11 illustrates the use of the active contour model on an MR brain image slice.

The tasks in medical image segmentation usually involve separating bones, tissue and blood. But it is also used for detecting specific structures, e.g. aneurysms and tumours. In the following section, a region-based method, known as Seeded Region Growing, is examined.

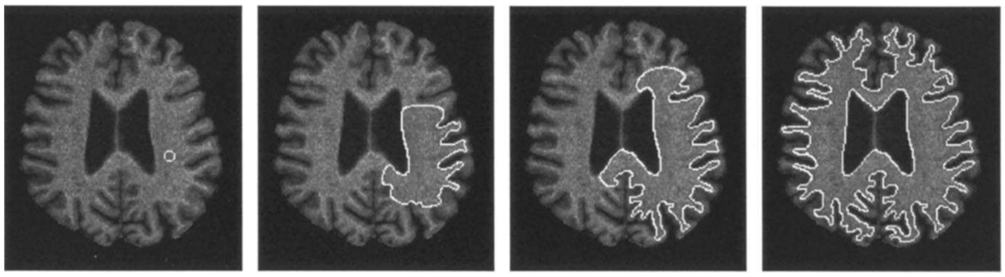


Figure 2.11: Snake segmenting gray-matter/white-matter interface and ventricles in an MR brain image slice. The initial contour is shown in the leftmost image. [23]

2.3.1 Seeded Region Growing

Seeded region growing was first introduced by Adams and Bischof in [1]. It is based on conventional region growing, where the general approach is to compare one pixel to its neighbours, and if a homogeneity criterion is satisfied, the pixel is said to belong to the same class as one or more of its neighbours. However, the mechanism of seeded region growing is closer to that of watershed segmentation.

The Basics of the Algorithm

The algorithm starts off with a set of points, known as seeds. These seeds are grouped into n sets, $A_1, A_2, ..., A_n$. From these sets, the regions inductively grow, and for each step of the algorithm, one pixel can be added to one of the sets. We look at the state of the sets A_i after m steps, and let T be the set of unallocated pixels that border with one or more of the regions.

$$T = \{x \notin \bigcup_{i=1}^{n} A_i | N(x) \cap \bigcup_{i=1}^{n} A_i \neq \emptyset\}. \tag{2.1}$$

If $x \in T$ and $N(x)$, the immediate neighbours of x, meets just one set A_i, then we define $i(x) \in \{1, 2, ..., n\}$ to be that index such that $N(x) \cap A_{i(x)} \neq \emptyset$. $\delta(x)$ is then defined as how different x is from the region it adjoins:

$$\delta(x) = |g(x) - \underset{y \in A_{i(x)}}{\text{mean}}[g(y)]| \tag{2.2}$$

where $g(x)$ is the gray value at the point x. If $N(x)$ meets more than one region, then $i(x)$ is set to the i that minimizes $\delta(x)$. Accordingly, a $z \in T$ is taken so that

$$\delta(z) = \min_{x \in T} \delta(x) \tag{2.3}$$

and this z is added to $A_{i(z)}$, and step $m+1$ is thus completed. Alternatively, the x can be classified as a boundary pixel, and added to a set of already-found boundary pixels, B, for display purposes.

The algorithm runs until all the pixels have been allocated, and Equation (2.2) and Equation (2.3) give a segmentation with homogeneous regions that fulfil the connectivity constraint.

A pseudo-code is given in Algorithm 1. It uses a sequentially sorted list (SSL), i.e. a linked list of objects, that stores the pixel adresses of T, and sort the elements by their δ-value.

Algorithm 1 Seeded region growing using boundary flagging.

```
Label seed points
Put the neighbours of seed points on the SSL
while SSL is not empty do
  Remove the first point $y$, from the SSL
  if all neighbours of $y$ which are labeled have the same label (not boundary
  label) then
    Label $y$ with the same label
    Update this region's mean value
    Add the neighbours of $y$ which are neither already set nor already in the
    SSL to the SSL according to their value of $\delta$
  else
    Flag $y$ with boundary label
  end if
end while
```

2.4 Related Work

Although we touch upon a relative new area of research, there exist some recent publications that are highly relevant to our work. Various filtering techniques and segmentation algorithms have been implemented successfully on the GPU, some achieving great speed-ups with respect to time consume, compared to equivalent CPU implementations.

2.4.1 Filtering on the GPU

Filtering is a major part of the visualization pipeline and is broadly used for improving images, reducing noise and enhancing detail structure. For example, low pass filters can reduce the noise of sampled medical volume images, and high pass filters can be used for edge extraction.

Linear filtering

Hopf et al. have implemented 3D convolution using graphics hardware in [17]. They take advantage of separable convolution kernels, and combine a 2D and a 1D convolution kernel. This means that the implementation consists of a 2D convolution on each slice in the volume, followed by a 1D convolution in the remaining direction.

Hopf et al. also show how hardware based wavelet filtering can be done by mapping the computations onto the OpenGL graphics pipeline in [18, 20]. Wavelet decomposition decomposes a set of data into a set of wavelet coefficients by using a mother wavelet, e.g. the Daubechies wavelet or the Haar wavelet. On the other hand, wavelet reconstruction recreates the data set from the coefficients. Wavelet decomposition and reconstruction are often implemented by applying convolution and down- and up-sampling steps to the volume data. As we have seen, graphics hardware is able to perform convolution, and together with the ability to scale bitmaps by arbitrary factors, we have what we need for the implementation. The fact that graphics hardware has memory systems that can be addressed very fast, is something that wavelet analysis can benefit from, since it for the most is a memory bound problem.

Hadwiger et al. present a general approach for hardware filtering in [14, 15]. They represent the different filter kernels as textures, and not as colour values, which allows filtering with arbitrary high-resolution kernels.

Nonlinear filtering

Preprocessing volume data by means of morphological operators using graphics hardware were performed by Hopf et al. in [19]. Unlike some filters that are based on linear combinations of the input data, these operators do not flatten the contours of the original data. The morphological operators map well onto the graphics hardware, since the per-fragment operations can perform maximum and minimum blending in the framebuffer. Hopf et al. decompose the structuring element into one-dimensional filters. This makes it possible to run through the algorithm in three passes, one for each direction.

Viola et al. introduce hardware-based nonlinear filtering in [30]. They have implemented median filtering, bilateral filtering and rotating mask filtering. Due to hardware limitations, their median filtering approach differs from histogram based approaches. They perform a binary search on the voxel values within a 5×5×5 filter mask to find the median. The bilateral filter is much like convolution-based smoothing, except that the contribution of voxels with values that significantly differ from the centre voxel's value is eliminated. Viola et al. implemented the Gaussian case of the bilateral filter. The rotating mask filter divides the operator mask into sub-regions, and the mean and the dispersion are calculated for each of these regions. The output value of the voxel is then set to the mean of the sub-region with the lowest dispersion. The authors used six different sub-regions. To reduce the number of texture fetch instructions, they decomposed their initial single-pass approach into six passes which increased the performance by a factor of 3.29.

Results

Overall, the authors of the different papers report good results. Some noticeable artifacts are mentioned in [17] due to that pixel fragments read from the framebuffer are clamped to [0,1] before they can be written back to the framebuffer or into the texture memory. These artifacts disappeared completely when using post-convolution and bias to map the expected results to the interval [0,1] just before the clamping takes place. In [19], the authors report no precision loss at all, although one might expect this due to limited framebuffer depth. This is because the morphological operations only use integer operations and the range of the results does not exceed the domain. However, the limited framebuffer depth does have an impact on the results in [18, 20]. The results showed only single bit errors in images of size 512^2, when complete wavelet decomposition and reconstruction were applied, using a framebuffer with a depth of 12 bits per base colour. With only 8 bits per base colour, the loss of accuracy was much greater. The hardware-based implementations of the mentioned filtering techniques consume less time than their equivalent software implementations. A 3D convolution on a data set of size 256^3 with a kernel of size 5^3 is approximately 5.88 times faster in hardware than in software according to [17]. In [18, 20], Hopf et al. report on filter times that differ with factors up to 5.2 and 2.8 for 2D wavelet reconstruction and decomposition respectively. By performing three different morphological operations in one cycle, filtering speed-ups of 15 times and more are reported in [19]. The median filter of Viola et al. in [30] shows a speed-up by a factor of 1.97, while the bilateral filter and the rotating mask filter have speed-up factors of 1.52 and 7.26 respectively.

2.4.2 Image Segmentation on the GPU

When it comes to using the GPU for segmentation purposes, the main focus so far has mostly been at level-sets and threshold based methods.

Threshold Based Segmentation

In [30], Vioala et al. have implemented simple thresholding on pre-filtered data sets. They found that this method produce an acceptable result for particular segmentation purposes, e.g. segmentation of the vessel structure of the liver. Compared to a software version, their segmentation algorithm on graphics hardware turned out to be 8.73 times faster. They also tried to send the segmented data back to main memory in compressed form. This increased the time in graphics hardware. However, it decreased the transfer time back to main memory with a factor of 20 and did not change the segmented result significantly.

Like Viola et al., Yang et al. have performed thresholding on graphics hardware in [32]. This is followed by morphological operations to smooth the object boundaries and remove spurious pixels. These two steps were implemented completely on the GPU by means of register combiners and blending technology. The register combiners were used when calculating square differences between pixels, and the results were written to the alpha channel. Then the alpha test was used to perform pixel thresholding. The erosion was implemented by setting the output pixel to the minimum of the corresponding input pixel and its eight neighbouring pixels, correspondingly the dilation used the maximum value for the output pixel. The hardware-based implementation proved to be over five times faster than a software version.

A Segmentation Method Based on Seeded Region Growing

Another presentation of a hardware-based segmentation method of structures from measured volume data is given by Sherbondy et al. in [26]. The algorithm is based on seeded region growing with a merging criteria that is based on Perona and Malik nonlinear diffusion metric. This criteria made the seeds merge into regions of similar intensity but with slower diffusion into voxels against high gradients. First, the user inserted the seeds, and a visualization of the calculations was given as the algorithm progressed. This made it easy for the user to see how the parameters affected the result, and the visualization was also efficient because the results were already in the video memory. Sherbondy et al. also made the algorithm even more efficient by using a computational masking technique. For each pass of the algorithm they only looked at a subvolume that contained all the voxels that had a chance of getting entered by a seed during that pass. This subvolume was obtained by a dilation of the segmented volume in each direction. The authors report on high performance gains, even without the computation masks, and compared to an optimzed CPU-based solution the algorithm ran 10-20 times faster.

Level-Set Implementations

A general-purpose segmentation tool that relies on interactive deformable models implemented as level-sets, is described by Cates et al. in [5]. This software application, called *GPU-based interactive segmentation tool*, GIST, is applicable to commodity graphics cards. Cates et al. map a 3D level set solver onto the GPU. This PDE[6] solver can give immediate respons to the user on the parameter settings, because it computes level set surface models at interactive rates. A region based speed function makes sure that the model grows into regions where the data is consistent with the desired segmentation and to contract in regions where it is not. This allows the user to intuitively specify the behaviour of the deformable model. The level-set solver in GIST achieved a speedup of a factor of 10-15 times over an optimized CPU-based solver.

Figure 2.12 shows a rendering of a cortical brain surface segmentation from a 256×256×175 MRI volume using GIST. The complete segmentation required no preprocessing of the data and took 5 minutes using a small, spherical surface (placed by the user) as the initial model. This type of segmentation is impractical to compute on ordinary, CPU-based solvers because of the size and complexity

[6]Partial differential equation

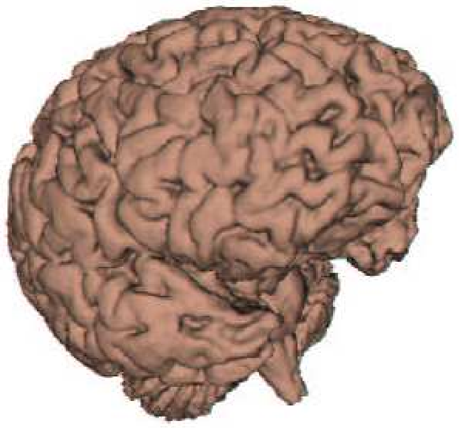


Figure 2.12: A segmentation of the cerebral cortex from a 256×256×175 MRI volume. [5]

of the solution. Compared with ITK[7], the same cortical segmentation typically takes more than an hour.

Level set segmentation using graphics hardware is also found in [25] by Rumpf et al.

2.5 CustusX

To help surgeons plan and perform minimal invasive surgical procedures, research scientists at Sintef are developing a navigation system called CustusX. In a close collaboration with radiologists and surgeons, the developers create a system based on techniques from navigation, 3D visualization and advanced medical image processing. The system as a whole consists of a dual processor, an optical position sensing system, and the actual software which is cross platform compatible. The surgeons are able to control the images with the surgical instruments so that the instruments are viewed in relation to the anatomy. CustusX may also be used for post-operative controls and educational purposes.

The segmentation of the volume data is performed by a module in the system arichitecture that uses the open source libraries of ITK. The segmented data are rendered into a 3D scene where it is possible to set the opacity of each of the segmented objects. The visualization is generated by means of the Visualization Toolkit[8] (VTK) from Kitware Inc.

[7]Medicine Insight Segmentation and Registration Toolkit, http://www.itk.org.
[8]http://www.vtk.org

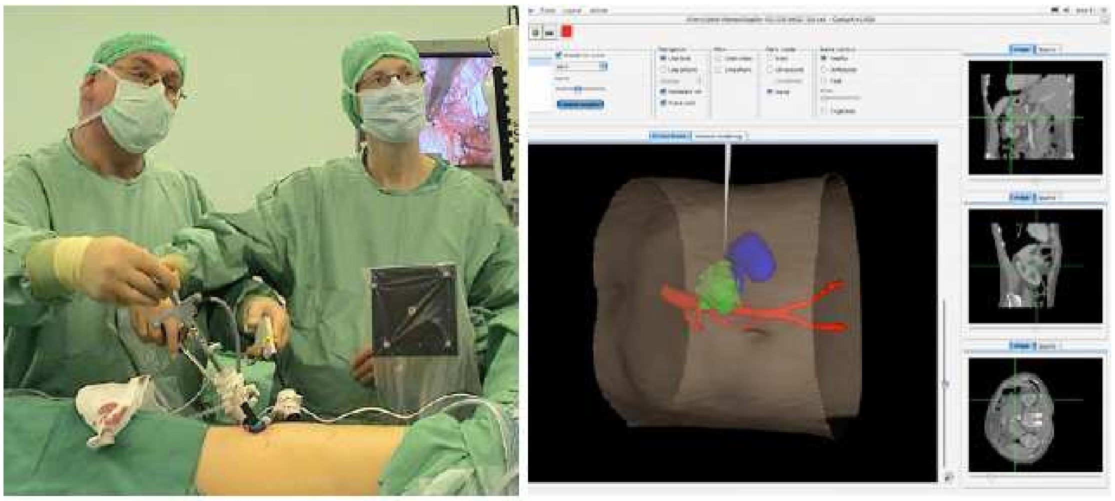


Figure 2.13: Surgeons in action using CustusX. Image screen to the right. [28]

Chapter 3

Design

3.1 Hardware Platform

With the enormous computation power of modern graphics hardware, and the fact that this growth has exceeded Moore's law, optimal performance is reached by utilizing the current state of the art graphics card. While developing our application, we will run the system on an Intel Pentium M 1.4 GHz processor with 512 MB of RAM using a NVIDIA GeForce FX Go 5200 64 MB graphics card. The final tests will run on an AMD XP 2200+ CPU with 512 MB RAM and a NVIDIA GeForce 6200 128 MB card. This GPU is capable of 128 pixel shader operations per clock cycle, and is the newest hardware series from NVIDIA. The vertex and fragment profiles used are the *OpenGL NV_vertex_program3* (vp40) and *OpenGL NV_fragment_program3* (fp40) respectively. They are new profiles introduced in the Cg 1.3 version.

3.2 Software

In this section we present the technology and software methods we are exploiting in our graphics application, and restrictions imposed on the design of the GPU seeded region growing system.

3.2.1 Graphics Library

Cg supports both the DirectX and OpenGL APIs. Even though OpenGL is known for its large overhead in render-context switching, which especially is an issue when using P-Buffers [31] for off-screen rendering, we use this API to manipulate the graphics state in our system. In addition, OpenGL is a cross-platform API whilst DirectX is a Microsoft Windows specific library. It is desirable that our graphics application will run under other operating systems than Windows XP, in particular under Mac OS. In our implementation, we use a new OpenGL extension, which is described in Section 3.2.3, and at the time of writing this is only supported in the latest Windows display drivers from NVIDIA.

3.2.2 A General GPU Framework

Sintef has developed a general GPU framework that incorporates both Cg and OpenGL Shading Language (see Section 2.2) for use with OpenGL. It wraps Cg functions and data, and offers simple functions for loading and initializing shader programs, and for enabling and disabling them when rendering graphics. We have used this framework with some minor modifications in our implementation.

3.2.3 Framebuffer Object Extension

When rendering graphics with OpenGL, a render-target must be used. This target determines the dimension and capabilities of the graphics pipeline. The number of channels (RGBA) present in the render-target determines how much information can be output per pixel. If the target has a depth buffer, the depth information can also be used as output.

Under normal display operations, the graphics card outputs the rendered graphics to the framebuffer. This buffer is allocated in video memory and is directly tied to what is seen in the application window on the screen. Often, you do not want the size of the output to be limited by the current window size.

Under several circumstances, it is necessary or useful to render to an off-screen buffer. This could be for creating dynamic textures, and other feedback effects such as procedural texturing and image processing. With off-screen rendering, *render-to-texture* can be used for creating dynamic texture data. As discussed in Section 2.1.3, calculated values after each rendering pass are often needed in the next pass. This could be overcome by copying data from the framebuffer back into texture memory, but this is very time-consuming, and hence the *render-to-texture* extension is desirable.

Pixel buffers (pbuffers) are widely used as off-screen render-targets. A pbuffer is a render surface with a set of properties chosen explicitly at run time. But this buffer consumes video memory and usually has its own OpenGL context, and therefore makes switching between pbuffers expensive. In addition, each pbuffer has its own depth, stencil and aux buffers. Thus, these buffers can not be shared between pbuffers. It is possible to bind the colour and depth buffer of a pbuffer as a texture (render-to-texture). This is window system specific, and portable applications need to create a separate pbuffer for each renderable texture.

A new[1] OpenGL extension called a *Framebuffer Object* (FBO), defines a simple interface for drawing to render-targets others than the buffers provided to the GL by the window-system. It only requires a single OpenGL context, so switching between framebuffers is faster than switching between pbuffers. Additionally, the format of a frambuffer object is determined by texture or renderbuffer format. One of the advantages in using this extension, is that renderbuffer images and texture images can be shared among framebuffers, e.g. share depth buffers between colour targets. This saves memory. It is also an easy task to switch render-target back to the window-system.

A framebuffer object is a collection of logical buffers: colour, depth, stencil and accumulation buffers. The render-targets are called *framebuffer-attachable images* and can be off-screen buffers (renderbuffers) and textures. The framebuffer object architecture is shown in Figure 3.1.

The state object defines where output of GL rendering is directed, and is equivalent to window system *drawable*. The renderbuffer contains a simple 2D image that stores pixel data resulting from rendering. The renderbuffer can be used for colour, depth or stencil information, dependent on the *attachment point*[2]. When a framebuffer object is bound, its attached images are the source and destination for *fragment* operations.

3.2.4 Image Format and Textures

The main challenge in our segmentation system is to perform segmentation on 3D data efficiently. The 3D data is read from a raw datafile, and each slice is uploaded to the graphics card as a 2D texture using the OpenGL function `glTexImage2D`. At the time of writing, rendering to 3D textures using the FBO extension is not

[1]Approved by ARB on January 31, 2005.

[2]State that references a framebuffer-attachable image. One each for colour, depth and stencil buffer of a framebuffer.

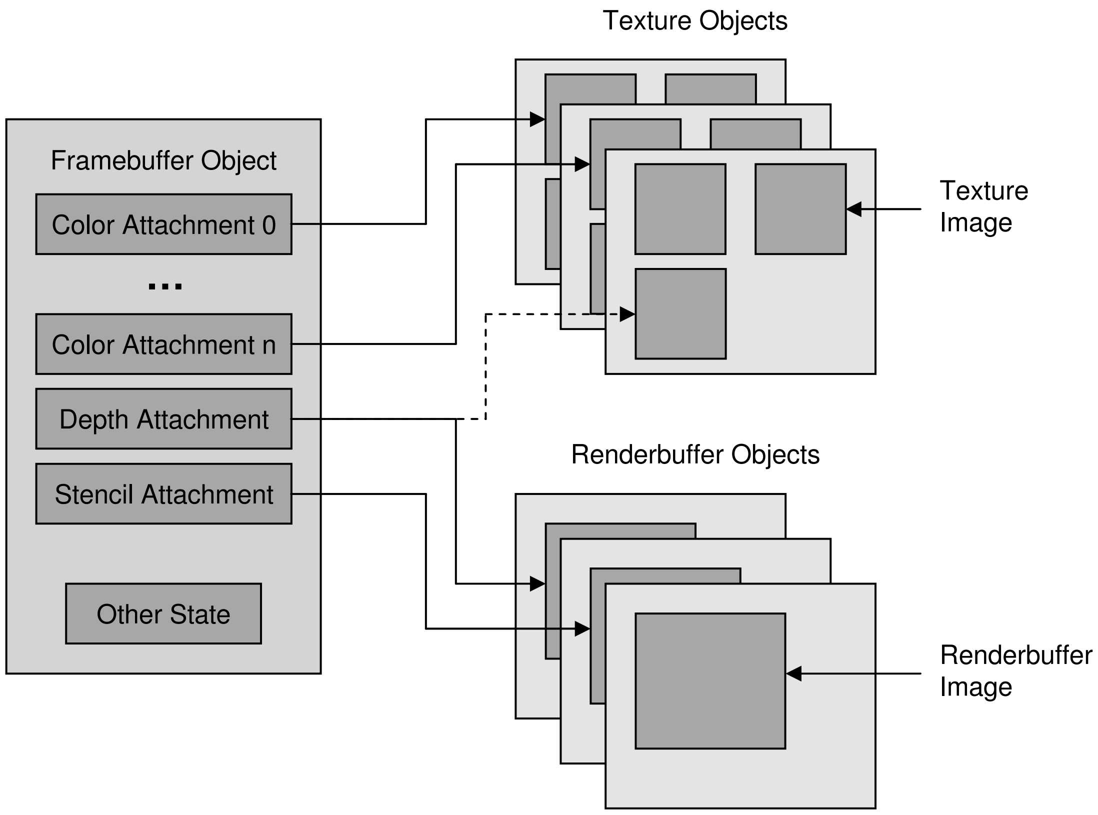


Figure 3.1: The Framebuffer Object architecture showing the logical buffers and attachable images [12].

supported in the graphics drivers provided by NVIDIA[3]. We are working with gray-scale images, and data in the input files are luminance values, each pixel consisting of one unsigned byte. The textures have RGBA format, and each pixel luminance value are copied to the RGB channels by means of the `glTexImage2D`[4] function, and attaching 1.0 for alpha. Similar yields for 2D images.

It is obvious that the texture data contains redundant information, since the luminance values are copied to each of the three RGB colour channels. As will be discussed in Section 3.4, only one channel needs to store the actual luminance value. The remaining three channels will be used for intermediate results and meta data in the segmentation process.

[3]The FBO specification includes the ability to rendering to 3D textures, but since this is an OpenGL extension, the graphics vendors may omit some of the functionality.

[4]The internal texture format `GL_RGBA` must be used in order to have four colour components in the textures. Using this internal format in conjunction with `GL_LUMINANCE` as pixel format, each luminance value is converted to floating point, and then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue, and attaching 1.0 for alpha.

3.2.5 Alpha-Testing

Alpha-testing is part of the OpenGL graphics pipeline. Based on a fragment's alpha value, a fragment is either discarded or it passes through this test stage. The application developer specifies an alpha test function and a reference value to which incoming alpha values are compared. If the comparison passes, the incoming fragment is drawn, conditional on subsequent stencil and depth-buffer tests. If the comparison fails, no change is made to the framebuffer at that pixel location. Figure 3.2 shows the different tests that can be enabled in OpenGL.

If fragments that are irrelevant to computation can be identified in advance and weeded out before they are processed by the fragment shader, it may result in a performance gain. An exclude shader will first be executed and exclude those fragments that are not wanted for further processing by setting an unique alpha value for those fragments. Later the alpha test function will discard them before a fragment shader executes.

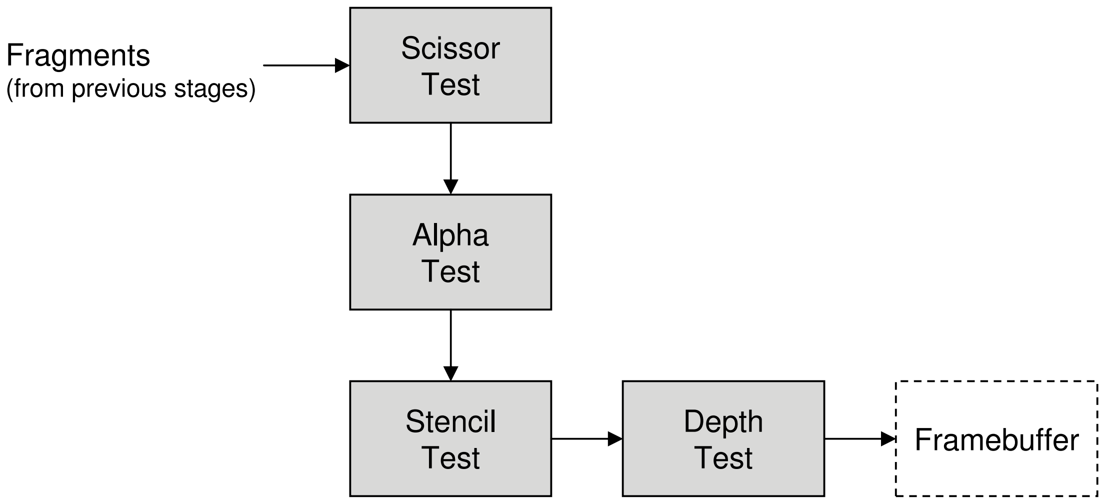


Figure 3.2: Tests that can be enabled in OpenGL, and which are applied to fragments in the graphics pipeline. If a test fails, no change is made to the framebuffer at that pixel location.

3.3 Filtering on the GPU

We want to map two different filtering techniques on the GPU for pre-processing our image data, i.e. median filtering and a nonlinear anisotropic diffusion filter. We start off with solutions for the 2D case which are easily extended for usage on 3D data.

3.3.1 Median Filter

Median filtering is usually done by taking an operator mask, e.g. a 3×3 mask, and setting the centre value to the median of the pixel values within this mask. However, the GPU has limited options for sorting values. Several conditionals are needed as well, something that is not recommended for usage on GPUs with respect to time consume. We therefore seek to implement an approximate version of median filtering. This version finds the horizontal median of the values within the operator mask, and then finds the vertical median. It does in some cases not find the correct median, but it guarantees, however, to find at least one of the two values closest to the median. This is generally good enough in most cases. The basis of the algorithm is depicted in Figure 3.3.

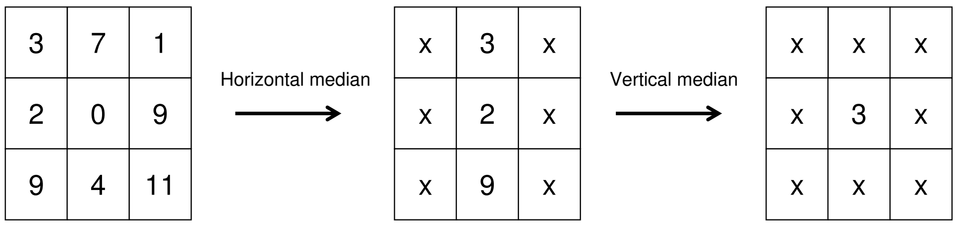


Figure 3.3: The approximate version of median filtering using a 3×3 operator mask. In this exampe the median should actually be 4, but 3 is one of the two closest values to the median. The **X**s are don't-care values.

For each fragment beeing processed, we need two additional texture lookups, the two closest neighbours. Then we must find the median of the neighbouring values and the center pixel. The algorithm must run in two passes, one for the horizontal median, and one pass for the vertical median.

3.3.2 Anisotropic Diffusion Filter

We base our nonlinear anisotropic diffusion filter on the method described in [11]. The advantage of this filtering technique is that it smooths within regions while preserving or sharpening the edges. The diffusive process can be formulated as in Equation (3.1).

$$\frac{\partial I(\mathbf{x},t)}{\partial t} = \nabla \cdot [c(\mathbf{x}, \nabla I(\mathbf{x},t)] \tag{3.1}$$

The vector $\mathbf{x}$ represents the coordinates, t is used for enumerating iteration steps, and $I(\mathbf{x}, t)$ is the image intensity. The strength of the diffusion is controlled by $c(\mathbf{x}, \nabla I(\mathbf{x}, t))$ which depends on the magnitude of the image gradient, and is given in Equation (3.2).

$$c(\mathbf{x}, \nabla I(\mathbf{x}, t) = \exp\left(-\left(\frac{|\nabla I(\mathbf{x}, t)|}{\kappa}\right)^2\right) \tag{3.2}$$

The κ, or *conductance*, is a parameter value that defines how much impact the gradient value will have on the smoothing process. Low values will give a better sharpening effect than with high values.

The algorithm is run in a user-specified number of iterations, and for the 2D case the pixel values will be updated according to Equation (3.3)

$$I(t + \Delta t) \approx I(t) + \Delta t \cdot \frac{\partial I(t)}{\partial t} \tag{3.3}$$

The integration constant, or time step, Δt, is user-defined and determines the iterative approximation of stability. There is no limitation for the lower bound of Δt. A small value results in a good approximation of the continous case, but requires many iteration steps. It can be shown that

$$\Delta t \leq \frac{1}{1+n} | c_i = 1, i \in \{1, \cdots, n\}, \tag{3.4}$$

i.e. the maximum time step in a 3D case using 6-connectedness is 1/7. The diffusive process is given in Equation (3.5).

$$\begin{aligned}
\frac{\partial}{\partial t} I(\mathbf{x},t) &= \operatorname{div}\left[c(\mathbf{x},t)\cdot \operatorname{grad}\, I(\mathbf{x},t)\right] \\
&= \nabla \cdot \left[c(\mathbf{x},t) \cdot \nabla I(\mathbf{x},t)\right] \\
&= \frac{\partial}{\partial x}\left[c(\mathbf{x},t)\cdot \frac{\partial}{\partial x} I(\mathbf{x},t)\right] + \frac{\partial}{\partial y}\left[c(\mathbf{x},t)\cdot \frac{\partial}{\partial y} I(\mathbf{x},t)\right] \\
&= \frac{1}{\Delta x^2}[c(x+\frac{\Delta x}{2},y,t)\cdot (I(x+\Delta x,y,t)-I(x,y,t)) \\
&- c(x-\frac{\Delta x}{2},y,t)\cdot (I(x,y,t)-I(x-\Delta x,y,t))] \\
&+ \frac{1}{\Delta y^2}[c(x,y+\frac{\Delta y}{2},t)\cdot (I(x,y+\Delta y,t)-I(x,y,t)) \\
&- c(x,y-\frac{\Delta y}{2},t)\cdot (I(x,y,t)-I(x,y-\Delta y,t))] \\
&= \phi_{east} - \phi_{west} + \phi_{north} - \phi_{south}. \qquad (3.5)
\end{aligned}$$

A pseudo-code for the implementation is given in Algorithm 2. For each fragment processed, the colour, i.e. gray-level value, is added a value calculated from the diffusive process multiplicated by Δt. Δx and Δy are set to 1 for simplicity, and the image gradients are calculated using the Sobel operators. The number of iterations is controlled by the user.

Algorithm 2 Pseudo-code for the 2D anisotropic diffusion filter. Shows how each fragment will be processed.

```
**Input** $texture$
**Input** $\kappa$
**Input** $\Delta t$

$gradient \leftarrow$ find gradient using Sobel operators
$c \leftarrow \exp(-(gradient/\kappa)^2)$
$\phi_{east} \leftarrow texture[x+1,y] - texture[x,y]$
$\phi_{west} \leftarrow texture[x,y] - texture[x-1,y]$
$\phi_{north} \leftarrow texture[x,y-1] - texture[x,y]$
$\phi_{south} \leftarrow texture[x,y] - texture[x,y+1]$
$diffusionTerm \leftarrow c * (\phi_{east} - \phi_{west} + \phi_{north} - \phi_{south})$

**return** $texture[x,y] + \Delta t * diffusionTerm$
```

The 2D anisotropic diffusion filter is easily extended to 3D. This is not described here.

3.4 Seeded Region Growing on the GPU

Our seeded region growing algorithm differs from the original that we looked at in Section 2.3. First of all we have to operate within the constraints of the GPU, and try to map the general algorithm onto the graphics hardware. The operations we perform must be generic for each fragment, and as seen in Section 2.1.3, we can only do write operations in relation to one fragment at a time, although we can perform many texture lookups. The algorithm must also have an iterative behaviour, since GPUs do not allow recursion.

Figure 3.4 shows an overview of our seeded region growing algorithm, where each rectangle represents a fragment shader. We start off with **Exclude** for excluding fragments with values that are not within a threshold interval. Then user-specified seeds are set, and these seeds grow into one or more regions. It is worth noticing that if two or more regions meet, they will automatically be merged into one region as we do not distinguish the regions from each other, i.e. we will use only one label for the initial seeds. After termination of the **GrowRegion**-shader, **ShadeSegmented** will take care of how the result will be displayed.

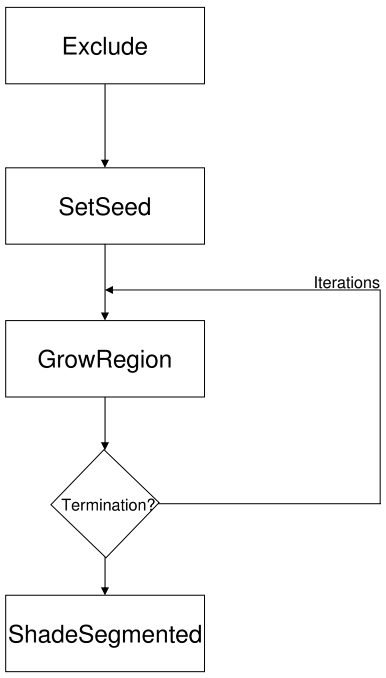


Figure 3.4: Overview of our seeded region growing algorithm.

The SRG algorithm is useful and will give good results when the intensity levels of the objects of interest are relative uniform. Additionally, the regions must not merge smoothly with other regions. Then, more sophisticated segmentation algorithms, like level-sets, must be applied. Anatomical structures like the aorta, some tumours, the cerebral cortex and the kidneys are examples of objects that are expected to be well segmented with the SRG algorithm. The liver, on the other hand, is releative hard to segment as it merges smoothly with other tissues.

The algorithm will follow the same scheme for both the 2D and 3D case. In the following sections, a description of each step is given.

3.4.1 Exclude

The user selects an upper and a lower threshold value so that only fragments with intensity values within this interval are considered by the region growing process. This shader will then go through all the fragments, and label those with intensity values not within the threshold interval as excluded. They do not need to be further processed.

3.4.2 Set Seed

This shader will label user-selected fragments as seeds. There is no option that allows for addressing an individual fragment, i.e. all of the fragments will be evaluated, and the texture coordinates will be matched with the user-specified input data. For the 3D case, however, one slice can be addressed alone, so we do not have to run through the entire volume data in order to set the seed points. Hence, only the slices that eventually will contain a seed point will be sent to this fragment shader.

3.4.3 Grow Region

This shader will make the seeds grow into one or more regions. Fragments with values that are not within the threshold interval have already been excluded for further processing, so basically what this shader has to do is to check if the current fragment has at least one seed as a 4- or 6-connected neighbour, for the 2D and 3D case respectively. If so, the current fragment is also labeled as seed. This is how fragments iteratively are labeled seeds and finally will constitute the segmented region.

We will make it possible for the user to define a tolerance value as well. As seen in Algorithm 3, this value can be used when new seeds are added. Then a new seed is only added if the difference between the current fragment's intensity value and a neighbouring seed's intensity value is less than or equal this tolerance value. This option can prevent adding fragments as seeds when their intensity values differ substantially from the values of their neighbouring seeds, although they are within the threshold interval. The 2D case is easily extended to three dimensions by adding two extra texture lookups, each in the texture slice above and below the current 2D texture.

To segment large regions or subvolumes, many iteration steps of the **GrowRegion**-shader are needed, but that also depends on the number of seeds that the user has initiated. More scattered seed points defined initially, will lead to faster convergence.

3.4.4 Termination

With respect to termination of the **GrowRegion**-shader, we will consider two different strategies. One option is to let the user specify the number of iterations. The other option will let the shader run until there are no more changes, i.e. no more seeds are added.

3.4.5 Shade Segmented

As some of the RGBA-channels are likely to be used for storing intermediate results, this shader is needed to display the segmentation result properly, as the visualizer and OpenGL use all three colour channels when rendering. The shader may also be used to shade the segmented data for increased realism of the visualization.

Algorithm 3 Pseudo-code for the **GrowRegion**-shader.

```
Input Texture $current$
Input Texture $above$
Input Texture $below$
Input Texture coordinates $(x, y)$
Input $\delta$

if $current[x-1, y]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - current[x-1, y])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if
else if $current[x+1, y]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - current[x+1, y])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if
else if $current[x, y-1]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - current[x, y-1])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if
else if $current[x, y+1]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - current[x, y+1])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if

  {Only for the 3D case}

else if $above[x, y]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - above[x, y])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if
else if $below[x, y]$ is a seed then
  $diff \leftarrow \mathrm{abs}(current[x, y] - below[x, y])$
  if $diff < \delta$ then
    Mark $current[x, y]$ as seed
  end if
end if

return $current[x, y]$
```

Chapter 4

Implementation

In Section 2.3.1 we described the basics of the Seeded Region Growing (SRG) algorithm and gave the mathematical fundamentals together with an algorithm showing how to merge unlabeled pixels to regions. We base our GPU implementation on the design given in Section 3.4, which try to map the original SRG algorithm to a GPU implementation. Concrete instances of **Exclude, SetSeed, GrowRegion** and **ShadeSegmented** are presented in detail in the following sections. Since each shader operates on a single fragment, and we are mainly dealing with 3D volumes, the shaders may process millions of fragments. And when iterating this to grow a region, it is essential to do optimizations in order to achieve acceptable results. We will focus our implementation along two axes: time and quality, the former beeing our main concern. The most important optmization is to remove conditionals, i.e. avoid `if`-statements. Loops are completely unacceptable. The different optimization techniques are discussed in the coming sections.

The programming language for the image segmentation system we are developing is C++, and Microsoft Visual Studio C++ .NET is used as developer environment. The GUI module of this system uses Qt[1], a cross-platform, open source C++ application development framework. Figure 4.1 shows a high-level UML class diagram of the most important components of the application.

The main part of the application, is the `ImageProc` class. This is the actual graphics class where all OpenGL calls are issued. The *Framebuffer Object Extension* (FBO), is used as off-screen render-targets. To be able to develop GPU programs in a high-level language, a Cg framework is implemented. The `ShaderLib` module is a general C++ shader framework written by Sintef, and was examined

[1]http://www.trolltech.com

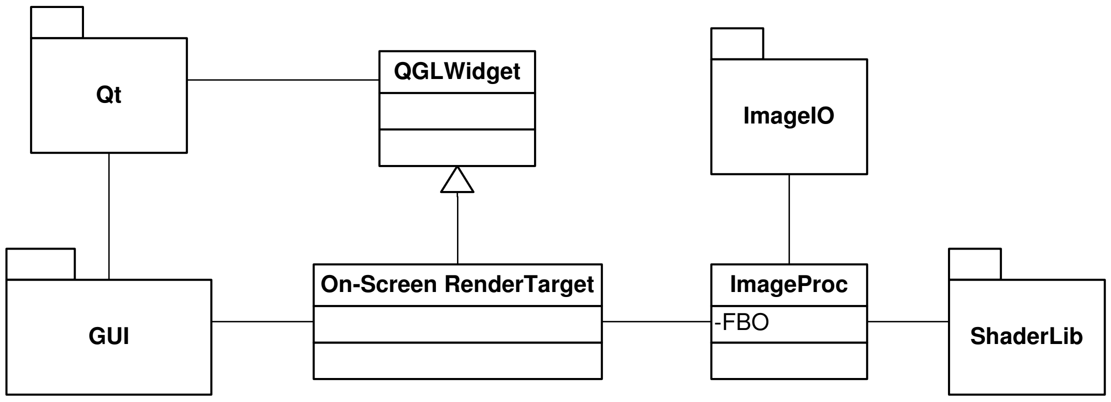


Figure 4.1: An UML class overview of the graphics application.

in Section 3.2.2. The shaders are loaded and compiled at run-time. All Cg programs developed are attached in Appendix A and B. The `QGLWidget` class is a Qt-class for rendering OpenGL graphics. Different image formats are supported by the application, and the loading and saving of image data is handled by the `ImageIO` module.

Dealing with 3D data, OpenGL 3D textures could be utilized. But the graphics card used under development and its driver did not support render-to-3D-texture, so this was not investigated. Instead, the volume is loaded as a set of 2D RGBA texture slices into texture memory. There are many reasons for using 3D textures in preference to 2D textures. The input data is 3D volumes, and when slicing a volume into 2D textures, the data may only be processed in one particular plane, e.g. the xy-plane. This makes it difficult to visualize the volume in all directions when blending is used. Additionally, it is often desriable to present for the user the slices in all three planes. Using 3D textures, we can simply access voxels by three-dimensional (x, y, z) coordinates, but using several 2D textures we first have to bind the texture corresponding to the z-coordinate[2], and then lookup texels with (x, y) coordinates.

The segmentation algorithm needs several user-defined parameters. This include seed points, threshold values and number of iterations. In response to this, we have developed a graphical user interface that makes this parameter setting easier. In addition, if the user is not satisfied with the segmentation result, it is an easy task to start from the beginning and tune the parameters. The GUI also makes it possible to investigate the segmentation result seen as a 3D rendered

[2]If each slice lies in the xy-plane.

volume.

When an input file (single slice or volume) is loaded, the OpenGL library is first initialized and then the image data are sent to the video memory using the `glTexImage2D` OpenGL call as described in Section 3.2.4. Subsequently, the framebuffer objects are allocated, and each 2D texture is attached to its own framebuffer object. All rendering is applied off-screen on one framebuffer object at a time, and the render output is written to the FBO's attached texture. Feeding the updated textures from one part of the segmentation process into the next step, this creates a "running" intermediate result that ultimately becomes the final result. The CPU side of the system is outlined in Algorithm 4.

All GPU programs are fragment shaders, since no vertex transformations are required. All shaders have at least two inputs and one output. These are the texture sampler (`samplerRECT`), the interpolated texture coordinates bound to `TEXCOORD0`, and the fragment's output colour bound to `COLOR`. `texRECT` is used to do texture lookups. The `texRECT` can be used with non-power-of-two textures. In OpenGL this is achieved by using `GL_TEXTURE_RECTANGLE_NV` as the texture target for all textures. After a Cg program is bound, each texture is mapped to a quadrangle (`GL_QUAD`) and rendered.

4.1 Excluding Fragments

We let the user define an upper and a lower threshold value, so that only fragments with values within this threshold interval will be further processed. As seen in Appendix A.1, the **Exclude**-shader checks if the `b`-component of the colour is within this interval. As we only work on gray scale images, we could have used the `r`- or the `g`-component, but the `r`-component will be used for intermediate results later. Hence, we may avoid some confusion by using the `b`- or `g`-component. The alpha channel, `color.a`, will be set to 1.0 (the maximum) if the value is within the threshold interval, and 0 if not, and at the same time all fragments are assigned the value 1.0 to the `r`-component. The reasons for this will become evident in the following sections.

Instead of using conditionals like the `if`-statement, we use Cg's `step`-function reffered to in Table 2.1 to directly calculate values. Since all meta data values have either a value of 0 or 1.0 (e.g. seed, not seed), this function is suitable to use because the output is either 0 or 1.0. So the fragment's `a`-component, `color.a`, is

Algorithm 4 Pseudo-code for the CPU side of the Seeded Region Growing system. As will be described in Section 4.5, alpha-testing can dramatically reduce the amount of fragments processed.

```
Input $lowerThreshold, upperThreshold$
Input Seed points

LoadImageData()
InitOpenGL()
UploadTextures()
InitFBO()

for all textures $t$ do
  BindFBO($t$)
  ExcludeFragments($lowerThreshold, upperThreshold$)
end for

for all seed points $(x, y, z)$ do
  BindFBO($z$)
  SetSeed($x, y$)
end for

EnableAlphaTest()
for all textures $t$ do
  BindFBO($t$)
  if $performOcclusionQuery$ then
    repeat
      GrowRegion()
    until no more fragments are rendered
  else
    for $i = 1$ to $numberOfIterations$ do
      GrowRegion()
    end for
  end if
end for
DisableAlphaTest()
```

assigned 1.0 if the fragment's intensity value is within the threshold interval, and 0 if not. Figure 4.2 shows the equivalent expression using `if`- and `else`-statements. There is no short-circuiting in Cg, so both sides of `&&` will always be evaluated.

```
//These expressions...
half f = step(lThreshold, color.b) * step(color.b, uThreshold);
color.ra = half2(1, f);

//...are equivalent to:
if(color.b >= lThreshold && color.b =< uThreshold){
        color.ra = half2(1, 1);
}
else{
        color.ra = half2(1, 0);
}
```

Figure 4.2: Optimization in the **Exclude**-shader, using the `step(a, x)` function

4.2 Seed Determination

The next shader to run is the **SetSeed**-shader. This shader receives the coordinates of seed points from the user and the corresponding fragments are labeled as seeds. This is done by setting the `r`-component of the fragment's colour to 0. Since we cannot address a fragment directly, this shader must check each fragment's coordinates in order to find out if it corresponds to a seed point. For the 3D case, however, we can address each slice individually so that we do not have to run through all the slices.

To see if a fragment has the matching coordinates with the user input, we had to do some adjustments. This was due to that the GPU operates on float values, so that a check for exact equality is not an option. Hence, we check if the fragment coordinates lie within ± 1 of the user-defined seed point coordinates.

The `r`-component, `color.r`, is used for labeling seeds. In the **Exclude**-shader, we saw that the `r`-component of all the fragments were initially assigned the value 1.0. So a fragment is labeled as seed by setting this component to 0. In this way we can easily distinguish between seeds and other fragments.

4.3 Grow Region

The **Exclude**-shader has set the alpha value to 1.0 if the fragment's gray level value was within the threshold interval. This makes it possible for us to only send the fragments that have the value 1.0 in the alpha channel to the **GrowRegion**-shader. A more thorough description of this process will be given in Section 4.5. Thus, we know that the fragments that are processed fulfil the gray value criterion. The shader now only checks if there are any seeds in a 4- or 6-connected neighbourhood in the 2D and 3D case, respectively. This is performed by multiplying the `r`-component of all the neighbour fragments. Since their `color.r`-value is either 0 or 1.0, the product of them will be 0 if at least one of them is a seed, and 1.0 if not. This value is then assigned to the current fragment's `r`-component indicating if it in this pass became a seed or not.

We implemented an alternative **GrowRegion**-shader as well. In addition to checking if at least one neighbour pixel is a seed, it also makes a constraint that the luminance value of the current fragment and a neighbouring seed do not differ with more than a user-defined value. The objective for this is to prevent the segmentation to leak through weak interfaces. As described in Section 2.3.1, the original SRG algorithm updates the mean of the corresponding region after a seed is added. This value then determines which region a new seed point will belong to, or if it is a boundary pixel. Implementing an updated region average value is more difficult on a GPU, as described in Section 2.1.3. Appendix A.3.2 shows the code for this shader. In comparison to Algorithm 3, we are avoiding a lot of unnecessary `if`-statements.

The execution of this shader must be iterated in order to discover large regions. A single pass will at most include six new seed points per seed point; four in the current slice, and one in the slice above and below. To prevent processing fragments that have already been labeled as seeds, we first check the `r`-value and discard the fragment for further proceessing if it is already a seed. As will be seen in Section 4.6, this can also help us deciding when the iteration loop should terminate.

4.4 Shade Segmented

This shader is run when the **GrowRegion**-shader has terminated, and is used for displaying purposes only. The user can decide if he wants to view the segmented regions in relation to the rest of the data or not. This is indicated by the variable `keepUnsegemented`, and if it is set to 1.0 the segmented regions are scaled in red and the rest is scaled in blue[3]. If it is set to 0, only the segmented regions will be displayed in gray scale. This shading is also neccessary to perform because of the meta data that is stored in some of the colour channels.

4.5 Computation Mask

As discussed in Section 4.1, the **Exclude**-shader excludes fragments that lie outside the given threshold interval. The **GrowRegion**-shader takes advantages of this as only fragments that lies within the threshold interval are processed. The observation that there are relatively few voxels in the volume that need to be computed using the **GrowRegion**-shader, leads to an optimisation. Using the alpha test described in Section 3.2.5 as a computation mask, the amount of fragments processed by the region growing shader can be dramatically reduced.

The **Exclude**-shader assigns the value 0 to the `a`-component of all fragments which luminance value does not lie within a user-defined threshold interval. Before the **GrowRegion**-shader executes, alpha-testing is enabled and the alpha test function only passes fragments whose alpha-value is 1.0. This is achieved through the OpenGL call `glAlphaFunc(GL_EQUAL, 1.0)`. The fragment shader will then only process potential seed points.

4.6 Termination

It is necessary to iterate execution of the **GrowRegion**-shader to let the region grow. The number of iterations can be user-defined, or the loop may terminate based on some criterion. Our application has both alternatives. Based on the region of interest, the user can specify the number of iterations that the **GrowRegion**-shader will iterate the volume (or slice in case of 2D). If this value is too conservative, the region will not be fully segmented. On the other hand, if the number of iterations is too many, time is wasted because the shader may execute even if no more seeds are added. The automatic approach is based on finding the amount of fragments that were actually rendered and written to the frambuffer in a rendering pass. OpenGL supports this through an *occlusion query.*

[3]This is because blending is performed individually on all three RGB-components.

As the name implies, this extension is normally used to quickly decide whether polygonal objects are visible and need to be rendered based on their mutual occlusions. The application can query the pixel count, i.e. number of fragments written to the framebuffer. If the returned value does not change between two consecutive rendering passes, this means that no further changes will occur henceforward, and the rendering loop may terminate. We have exploited the occlusion query extension in our implementation to automatically decide when to terminate the region growing. The Cg `discard` statement terminates execution of the program for the current fragment and suppresses its output. This is done if a fragment already is a seed. When no more seeds can be added, no changes are made in the frambuffer, and hence a constant number of fragments are written. This is detected by means of the occlusion query returning the same pixel count for two consecutive rendering passes. Algorithm 5 shows how this is performed.

Algorithm 5 Using occlusion query to terminate the region growing. The GrowRegion() discards fragments that are already seeds. When two consecutive rendering passes write the same number of pixels to the framebuffer, the loop is terminated.

```
$lastPixelCount \leftarrow 0$
$pixelCount \leftarrow 0$
GenerateOcclusionQuery($query$)

repeat
  $lastPixelCount \leftarrow pixelCount$
  BeginOcclusionQuery($query$)

  GrowRegion()

  EndOcclusionQuery($query$)
  $pixelCount \leftarrow$ GetOcclusionQueryPixelCount($query$)
until $lastPixelCount \neq pixelCount$
```

4.7 Filters

We have implemented the two filters described in Section 3.3, i.e. median filtering and a nonlinear anisotropic diffusion filter. We will now take a closer look at these implementations. The source code of the shaders is given in Appendix B.

4.7.1 Median Filter

The 2D median filter is implemented using two Cg shaders. To be able to perform median filtering on 3D data, we execute the 2D filter on each of the image slices. The first, `medianH`, finds the horizontal, one-dimensional median of three consecutive horizontal pixel values, and assigns the value to the centre pixel of the three. The following shader, `medianV`, then finds the vertical median in the same manner. This makes sure that each pixel will be assigned the median value within a 3×3 mask, only with the small errors discussed in Section 3.3.1. It is desirable to avoid using conditionals for calculating the median of the three intensity values. Inspired by sorting networks, we thus found the median using only `min` and `max` operations. Figure 4.3 illustrates this difference.

```
/*Finding the median of a, b and c*/

//Using only min and max operations...
median = max(min(a, b), min(max(a, b), c));
//Instead of writing...
if (a < b) {
    if (b < c) {
        median = b;
    }
    else {
        median = max(a, c);
    }
}
else {
    if (a < c) {
        median = a;
    }
    else {
        median = max(b, c);
    }
}
```

Figure 4.3: Optimization in the Median-shaders, using only `min` and `max` operations for calculating the median.

4.7.2 Nonlinear Anisotropic Diffusion Filter

The basics of this filter were discussed in Section 3.3.2. We started off with implementing the filter for the 2D case, and then extended it for usage on 3D data. The user specifies the number of iterations, the conductance κ and the time step, Δt. The last two are sent to the diffusion shader as a 2-vector uniform parameter, `k_step = float2(k, step)`. The shader is then executed in each iteration step, and the luminance value of each fragment is added the product of the time step, i.e. `k_step.y`, and the value returned from the diffusive process function, `dI`.

The `dI`-function calculates the $\phi_{direction}$ values as in Equation (3.5) by consecutive calls to the gradient function `grad` and the diffusion function `c`. The `grad`-function calculates the gradient value of the fragment using Sobel operators. The `c`-function calculates the diffusion strength according to Equation (3.2), using the gradient value and the κ value, i.e. `k_step.x`.

$$\mathbf{h_x} : \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \qquad \mathbf{h_y} : \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

Figure 4.4: The 2D Sobel operators.

The Sobel operators that we use for calculating the gradient values for 2D data are depicted in Figure 4.4. In the 3D case, we use the filter depicted in Figure 4.5 in each of the three directions. In order to get a scalar gradient value, we sum up the absolute value of the directional components.

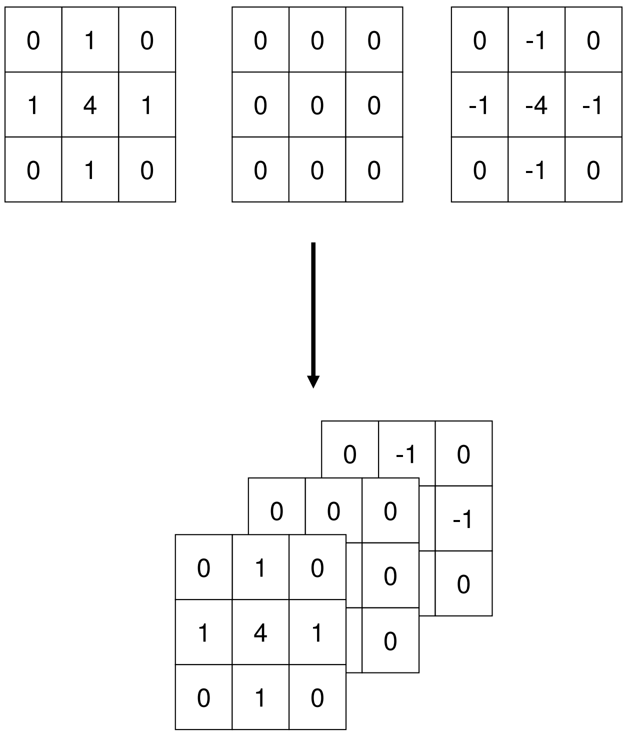


Figure 4.5: The 3D gradient operator, used for each direction

Chapter 5

Visualization

We have developed a volume renderer that is able to visualize a 3D volume in an acceptable way. The output of the Seeded Region Growing is also visualized as a 3D volume and makes the inspection of the segmentation result easier. In this chapter we investigate the techniques used to render 3D volumes and look on some examples obtained by our volume renderer. There exist a plethora of visualization methods in the literature. We base our implementation on a GPU based volume renderer using textures and blending that is easy to implement [27].

5.1 Volume Rendering

The method described in [27] makes use of 3D textures. But the technique is straightforward using 2D textures aswell. The volume data is first uploaded to the video card as 2D textures. Each slice is sampled from a particular plane, e.g. the xy-plane. Then multiple planes parallel to the image plane are used to sample the textures and sent to the geometry-processing unit. The GPU is exploited for interpolating the 2D texture coordinates provided at the polygon vertices and for reconstructing the texture samples by interpolating within the texture slices. In the final stage, the pixel values are blended into the framebuffer in order to approximate the continous volume integral.

While using 2D textures, every polygon vertex is given a point in texture space. The GPU maps the values from the texture onto the polygon surface by interpolating the texture coordinates. Each polygon is modeled as a square[1] and is drawn parallel to the projection plane in the screen space at different depths.

[1]GL_QUAD

Figure 5.1 depicts the stages in the volume visualization on the GPU.

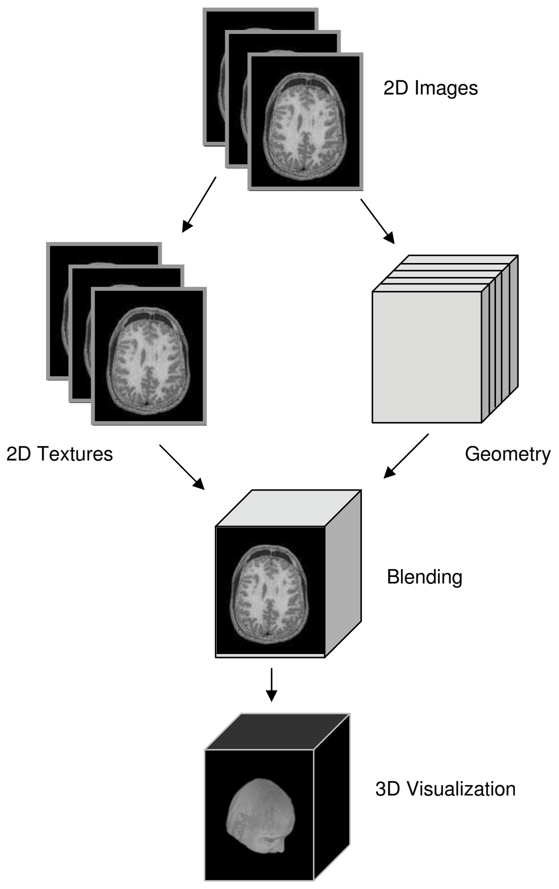


Figure 5.1: Stages in the volume visualization using 2D textures and blending on the GPU.

With volume rendering, it is possible to make the internals of the volume to be visible by assigning optical properties like colour and opacity to the voxel data. The opacity of a surface is a measure of how much light penetrates through the surface. An opacity of 1.0 ($\alpha = 1.0$) corresponds to a completely opaque surface that blocks all light incident on it. On the other hand, a surface with an opacity of 0 is fully transparent; all light passes through it. The transparency of a surface with opacity α is given by $1 - \alpha$.

If we want to use blending, we need a way to apply opacity as part of the rendering process. If we regard the fragment beeing rendered as the source pixel and the framebuffer pixel as the destination, we can combine these values in various ways. If we represent the source and destination pixels with the four-element RGBA arrays

$$\mathbf{s} = (s_r, s_g, s_b, s_a), \tag{5.1}$$

$$\mathbf{d} = (d_r, d_g, d_b, d_a) \tag{5.2}$$

then a compositing operation replaces **d** with

$$\mathbf{d}' = (b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a) \tag{5.3}$$

where the arrays of constants $\mathbf{b} = (b_r, b_g, b_b, b_a), \mathbf{c} = (c_r, c_g, c_b, c_a)$ are the *source* and *destination blending factors* respectively.

Since only the luminance is recorded in the original images, the fragments sent to the framebuffer do not have any opacity information. But, instead of mapping each luminance value to an opacity value, we use the colour information directly to perform blending. The source and destination blending factors are $\mathbf{b} = (s_r, s_g, s_b, s_a)$ and $\mathbf{c} = (1, 1, 1, 1) - (s_r, s_g, s_b, s_a)$ respectively. This means that black $(0, 0, 0)$ is fully transparent whilst white $(1, 1, 1)$ is completely opaque.

5.2 Rotation

The geometry is rendered from farthest to nearest with respect to the current orientation of the volume. Our visualizer supports rotation around all three axes, and hence care must be taken when rendering the quadrangles to perform blending in the right order. Figure 5.2 shows how the geometry is rendered and blended from farthest to nearest along the z-axis.

The order in which the geometry is rendered depends on the relative direction of the axis perpendicular to the image plane (z-axis) and the projection plane (screen). If the basis orientation of the object is as depicted in Figure 5.2 and the camera is oriented in origin pointing along the negative z-axis, we define the textures which are rendered farthest and nearest as tex_{far} and tex_{near} respectively. The depth interval is $[-z_{far}, z_{near}] = [-numOfSlices/2, numOfSlices/2]$. In short, tex_{far} is mapped to the quadrangle at $-z_{far}$ and tex_{near} is mapped to the quadrangle at z_{near}. But, if a rotation of $(\pm 90, \pm 270]$ degrees is carried out around the x- or y-axis, the order in which the textures are rendered, must be reversed. Tracking this shift in rendering order is not as straightforward as it may seem at

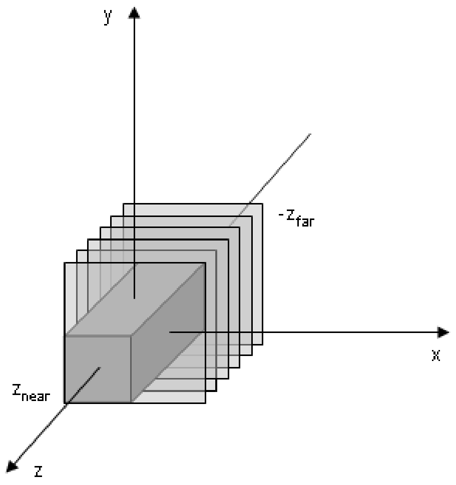


Figure 5.2: Rendering and blending a volume along the z-axis.

first glance. An arbitrary combination of rotations around the x- and y-axis may require to render in reverse order. Instead of directly tracking the rotations, we can look at the orientation of the axis through the object perpendicular to the image plane (the object's z-axis in model space) with respect to the projection plane. If we define a normalized direction vector $\mathbf{d} = (0, 0, 1)$ in model space, and let this go through the same transformations as the object being visualized, we can simply check the sign of $\mathbf{d}'.z$ where $\mathbf{d}'$ is the transformed vector. If $\mathbf{d}'.z \geq 0$ we have the situation depicted in Figure 5.2. Otherwise, if $\mathbf{d}'.z < 0$ the rendering order is reversed, and tex_{far} is mapped to the quadrangle at z_{near} and tex_{near} is mapped to the quadrangle at $-z_{far}$. Checking the sign of this z-component only require us to check the sign of the 11th element in the current model view transformation matrix. To see why, consider Equation 5.4 on how 3D primitives are transformed.

$$\begin{aligned}
\mathbf{p}' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} &= \mathbf{Mp} \\
&= \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
\end{aligned} \tag{5.4}$$

p' is the transformed point, **M** is the transformation matrix and **p** is the point in model space. Inserting the direction vector **d** into Equation 5.4 gives

$$\begin{aligned}\mathbf{d}' = \begin{pmatrix} x' \\ y' \\ z' \\ 0 \end{pmatrix} &= \mathbf{Md} \\ &= \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \alpha_{13} \\ \alpha_{23} \\ \alpha_{33} \\ 0 \end{pmatrix} \end{aligned} \tag{5.5}$$

Hence, we see that $\mathbf{d}'.z = \alpha_{33}$ is the 11th element of the model view transformation matrix **M**. In OpenGL, the model view transformation matrix can be fetched through a call to the OpenGL function `glGetFloatv` with `GL_MODELVIEW_MATRIX` as a parameter.

It is desirable to rotate the object in screen space, i.e. around a fixed coordinate system. The user controls the rotation by moving the mouse. There is no simple way to handle this in OpenGL, since each transformation actually changes the coordinate system of the model with respect to that of the camera. A transformation carried out later will then transform the object with respect to the object's local coordinate system, not the fixed screen or viewing coordinate system. What is needed, is a way to store the transformations carried out on the object, in particular the rotation transformations. Quaternions may be used for this, but in our implementation we use a more direct and simpler approach. The root cause of the problem is that OpenGL matrix operations postmultiply onto the matrix stack, thus causing transformations to occur in object space. To affect screen space transformations, we need to premultiply. OpenGL does not provide a mode switch for the order of matrix multiplication, so we need to premultiply by hand. We implement this by retrieving the current matrix after each frame. We then multiply new transformations for the next frame on top of an identity matrix and multiply the accumulated current transformations (from the last frame) onto those transformations. This is summarized in Algorithm 6.

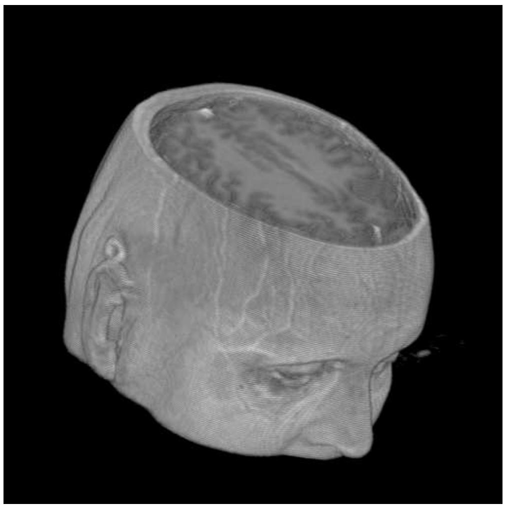
A problem when using 2D textures manifest itself when it comes to rotation. When viewing the volume such that each image slice is perpendicular to the viewing plane, the object will be less visible. In fact it will disappear when the slices are parallel to the xz-plane in world space. This is because each slice is rendered and blended along the z-axis. A solution to this is either to use three different texture sets sampled from all three different planes, or to utilize 3D textures.

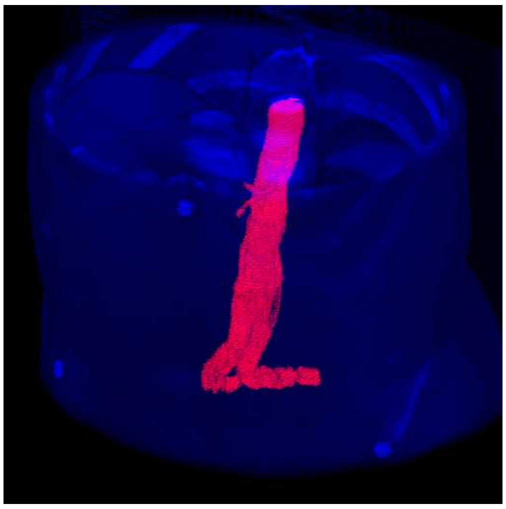
Algorithm 6 Rotation around a fixed coordinate system in OpenGL.

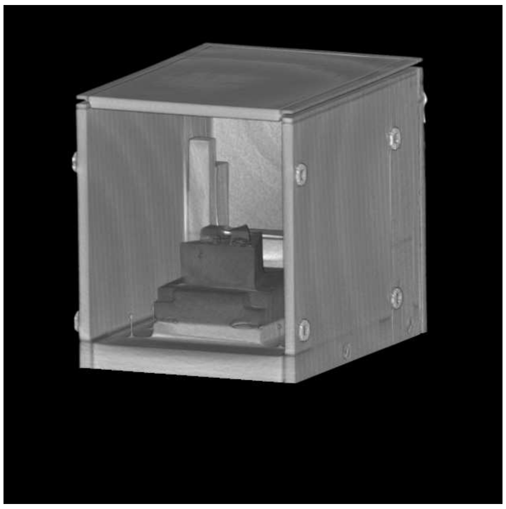
```
Require: $curMatrix$ {Current model view transformation matrix}

  LoadIdentity()
  PerformTransformations() {Changes the matrix on top of the matrix stack}
  MultiplyMatrix($curMatrix$) {Multiply matrix on top of stack with $curMatrix$}
  RenderGeometry()
  $curMatrix \leftarrow$ GetCurrentModelviewMatrix()
```

5.3 Examples

Figure 5.3 shows four volumes rendered by our application.


(a) Rendered volume of a human's head.


(b) A visualization of a segmented aorta.


(c) Rendered volume of phantom data.

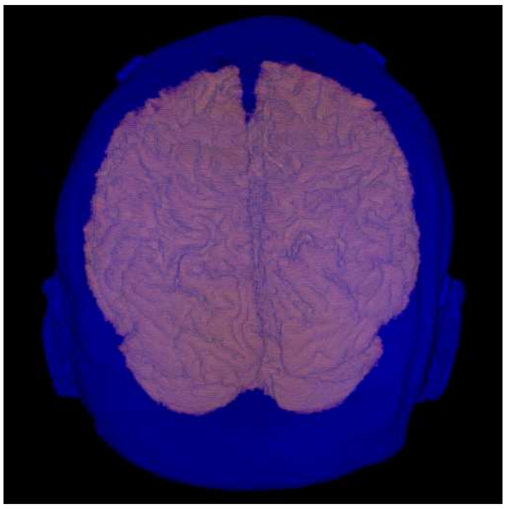

(d) Segmentation of the cerebral cortex.

Figure 5.3: Four volumes rendered by our graphics application.

Chapter 6

Results

This chapter gives a survey of how and under which conditions the segmentation results are achieved. This is followed by a presentation of the actual results.

6.1 Test Setup

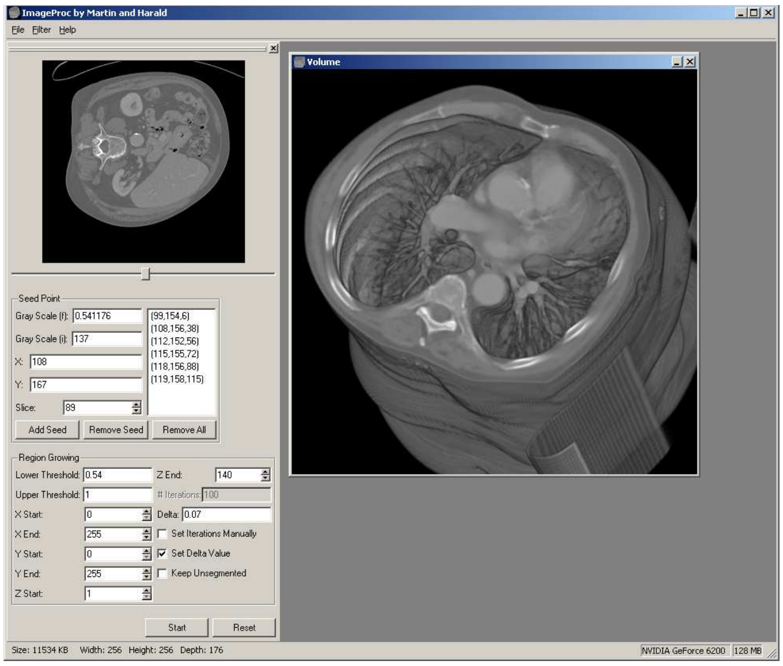
We start off by looking at the hardware platform of our test system. Then we take a look at the different volume data sets that we use, and which of the anatomical structures that are of interest to us. To round off this section, we mention what we will measure in our tests, and how these measurements are achieved.

6.1.1 Hardware Platform

All tests will run on an AMD XP 2200+ CPU with 512 MB RAM and a NVIDIA GeForce 6200 128 MB graphics card. This GPU is capable of 128 pixel shader operations per clock cycle, and is the newest hardware series from NVIDIA. The vertex and fragment profiles used are the *OpenGL NV_vertex_program3* (vp40) and *OpenGL NV_fragment_program3* (fp40) respectively. They are new profiles introduced in the Cg 1.3 version.

6.1.2 Graphical User Interface

The segmentation algorithm needs several user-defined parameters. This include seed points, threshold values and number of iterations. In response to this, we have developed a graphical user interface that makes this parameter setting easier. In addition, if the user is not satisfied with the segmentation result, it is an easy task to start from the beginning and tune the parameters. The GUI also makes it possible to investigate the segmentation result seen as a 3D rendered volume. Figure 6.1 shows a screen shot of the GUI.



Gray Scale (f): 0.541176
Gray Scale (i): 137
X: 108
Y: 167
Slice: 89

(99,154,6)
(108,156,38)
(112,152,56)
(115,155,72)
(118,156,88)
(119,158,115)

Lower Threshold: 0.54
Upper Threshold: 1
X Start: 0
X End: 255
Y Start: 0
Y End: 255
Z Start: 1
Z End: 140
Iterations: 100
Delta: 0.07
☐ Set Iterations Manually
☑ Set Delta Value
☐ Keep Unsegmented

Size: 11534 KB Width: 256 Height: 256 Depth: 176 NVIDIA GeForce 6200 128 MB

Figure 6.1: A screen shot of the GUI.

6.1.3 Test Data Sets

We have received data material from Sintef Health Research for testing our GPU implementation of seeded region growing. The data sets reveal anatomical structures within the abdominal area and the head, and were captured using Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) scanners respectively. Table 6.1 gives a complete overview of our data sets. Each volume has been given a unique identifier in order to distinctively address each data set. The size of the volume is also given, both in the various dimensions and with respect to the number of voxels. This becomes relevant when measuring the efficiency.

6.1.4 Relevant Structures

We will focus our segmentation on some anatomical structures in the different sets of medical volume data. In the abdominal area it will be the aorta, the liver, and

Set ID	Set Type	Acquisition Method	Volume Size	Number of Voxels
A1	Abdomen	CT	256×256×350	22937600
A2	Abdomen	CT	256×256×176	11534336
A3	Abdomen	CT	256×256×176	11534336
A4	Abdomen	CT	256×256×129	8454144
H1	Head	MRI (T1 weighted)	256×256×179	11730944

Table 6.1: The data test sets that we use.

the kidneys that we will concentrate on. We also want to test our implementation on a brain tumor and the cerebral cortex from the volume data of the head.

6.1.5 Achievement and Evaluation of the Results

In [29], Udupa et al. suggest to focus on three factors for evaluating a segmentation algorithm, namely efficiency, accuracy and precision. When evaluating our implementation, the main attention will be to the time consume of the algorithm. This is because we are not interested in checking the overall performace of the algorithm per se, but merely in the efficiency of implementing it on the GPU.

Udupa et al. also argue that both human operator time and computational time need to be considered when characterizing the efficiency. We will focus on the computational time, i.e. the time consume of an execution of the algorithm. So the metric for efficiency will in our case be the time it takes for the GPU to complete its computational tasks.

The time consume will be measured using various input. First of all, the size of the volume data will have a large impact. However, the number of initial seed points and their position will also affect the computation time, as a high number of scattered seeds leads to faster convergence. The same goes with the threshold interval, as it defines how many fragments that are excluded from further processing. Finally but not least, using different number of iterations will of course affect the total time consume.

The time will be measured using the Windows specific `_ftime` function. The total time spent in hardware will be calculated as depicted in Figure 6.2. In this way we will try to count the actual processing time on the GPU, but some overhead is included due to shader and render-target (FBO) setup.

For a comparison, we will do some equivalent tests on a CPU implementation of seeded region growing. The CPU version will be equivalent to our GPU imple-

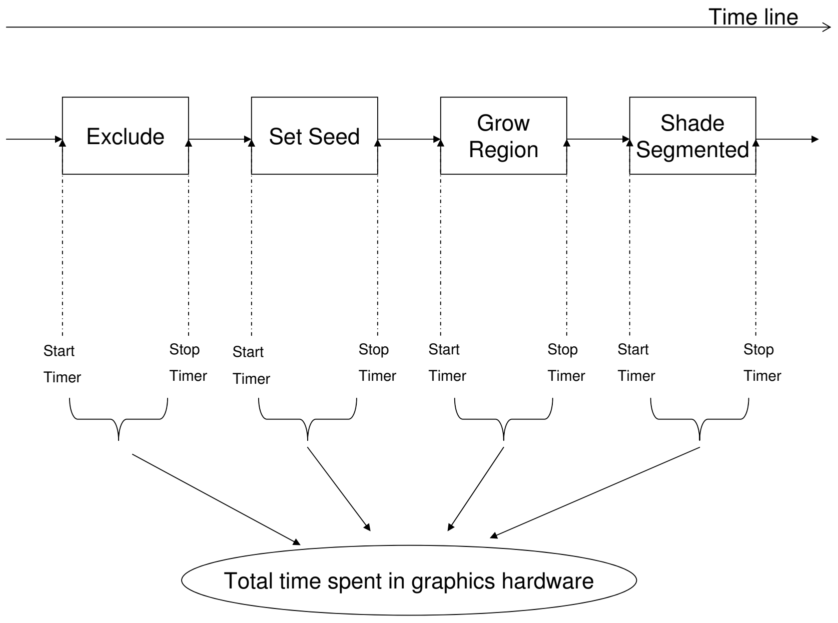


Figure 6.2: Measuring total time spent in hardware.

mentation in the way that voxels are processed. Thus, the CPU version will have an iterative behaviour and process all voxels. This is different from the traditional, recursive implementation.

6.2 Results

In this section we present the results of running our GPU based SRG algorithm. We will use different input volumes in order to segment out the various structures mentioned in Section 6.1.4. All segmentation results obtained for each structure are covered in the following sections. In Table 6.2, the parameters used in all segmentation tests, in addition to time consume, are given. In Section 6.4 visualization of the results are shown.

6.2.1 Aorta

The aorta was segmented in all abdominal data sets. This structure is quite easy to segment using the SRG algorithm because of the relative uniform intensity levels. In addition, the aorta does not smoothly merge into other regions, but forms a distinct region separated from the surrounding matter. A slice from data

Set ID	Structure	Seed Points	Threshold Interval	Delta	Slices	Itera-tions	Time (ms)
A1	Aorta	9	[0.57, 1.0]	-	1-260	50	9563
A1	Aorta	9	[0.57, 1.0]	-	1-260	74 [1]	16613
A1	Aorta	1	[0.57, 1.0]	-	1-260	200 [2]	44654
A2	Aorta	9	[0.6, 1.0]	-	30-174	50	4186
A2	Aorta	9	[0.6, 1.0]	0.05	30-174	50	6390
A3	Aorta	10	[0.54, 1.0]	-	1-140	50	4065
A4	Aorta	5	[0.7, 1.0]	-	1-127	50	4667
H1	Brain Tumour	4	[0.45, 0.85]	0.035	80-110	30	926
H1	Cerebral Cortex	8	[0.45, 1.0]	0.05	40-152	80	9253
A2	Liver	8	[0.43, 0.56]	0.04	105-167 [3]	35	911
A4	Kidneys	2	[0.495, 1.0]	-	1-129	55	5118

Table 6.2: Segmentation settings and achieved computation times.

set A1 is shown in Figure 6.3. No preprocessing was required to achieve good results. Except for data set A2, no delta value was used. Segmenting this data set using the ordinary SRG algorithm resulted in over-segmentation, where part of the spinal marrow was erroneously included in the segmentation result. This is caused by the weak interface between the aorta and the spinal marrow. A delta value of 0.05 was used to correct this.

As seen in Table 6.2, using more seed points, the algorithm leads to faster convergence. Row 2 and 3 prove this. In these two experiments, the segmentation settings, except the number of seed points, were identical. Using occlusion query to terminate the region growing, the time spent to converge using only a single seed point, was almost three times the time using 9 seed points. Faster convergence is also achieved when the seed points are more scattered.

6.2.2 Brain Tumour

The brain tumour in data set H1 was a hard task for our algorithm. The voxels within the tumour had gray values ranging from approximately 0.45 to 0.85. This is a large interval, and the surrounding voxels of the tumour have similar intensity

[1]Occlusion Query was used.

[2]Occlusion Query was used.

[3]Volume also narrowed down in the x- and y-coordinate, from 29 to 170 and 79 to 215, respectively.

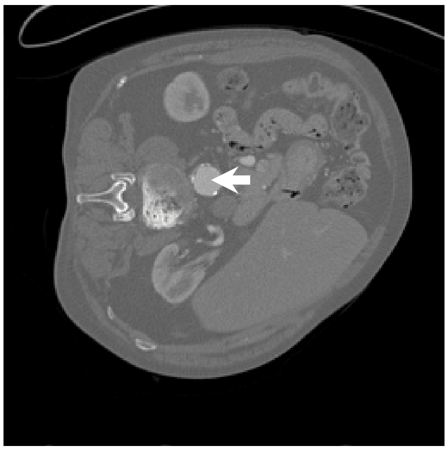
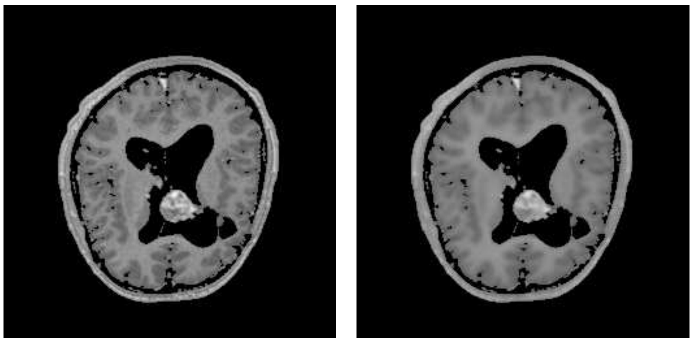


Figure 6.3: Slice from data set A1 showing the clear distinction of the aorta with surrounding matter.

values. Hence, to prevent the seeds from growing outside the tumour, we had to make some restrictions.

The head in the data set was surrounded by voxels with low gray level values. Hence, we first did some basic thresholding on the original data in order to eliminate these voxels. We used a threshold value of 0.35. In order to smooth the gray values in the tumour while preserving the edges, we used our anisotropic diffusion filter with a κ-value of 0.2, time step of 0.1, and 10 iterations. This effect is shown on one of the image slices in Figure 6.4.



(a) Input image slice. (b) Filtered image slice.

Figure 6.4: Result of using the anisotropic diffusion filter with a κ-value of 0.2 and with 10 iterations.

When using occlusion query for termination of the algorithm, we found that the seeds grew into the matter outside the tumour. To prevent this from happening, we specified a number of iterations that the algorithm was allowed to run. In addition, we only considered a subvolume of the head, instead of the entire volume. The constraint was in the z-direction, i.e. we specified which slices we wanted to look at.

The algorithm leads to faster convergence when using many, scattered seed points. We started with four seed points, all placed within the tumour, but within different image slices. We used a delta value of 0.035 to prevent seeds growing into voxels with values much different from their own.

Figure 6.9 shows the resulting volume and the segmentation of one of the image slices. There are some spurious holes within the segmented part of the slice. This is most likely due to that the delta value was used. However, without it a much less distinct segmentation of the tumour would occur.

6.2.3 Cerebral Cortex

The cerebral cortex was successfully segmented using our SRG implementation. This structure was quite sensitive to the input parameters, and most tests resulted in over-segmentation where part of the cranium was erroneously included. The final segmentation was achieved by a preprocessing step consisting of anisotropic diffusion filtering followed by the SRG using a small delta value. Parameters for the diffusion filter was $\kappa = 0.07$, $\Delta t = 0.1$, and 5 iterations. Number of iterations for the region growing was manually set to 80. We also tested the segmentation using occlusion query. This resulted in 318 iterations and over-segmentation.

6.2.4 Liver

It is complicated to isolate the liver using the seeded region growing algorithm, and much more complex methods are actually needed in order to get acceptable results. Although the intensity levels of the voxels are similar whithin the region, much of the surrounding voxels also have approximately the same distribution of intesity levels. We did, however, put our implementation to this difficult test.

We used data set A2 and performed some pre-processing consisting of anisotropic diffusion filter with $\kappa = 0.07$, $\Delta t = 0.1$, and 5 iterations. Then we inserted 8 seed points into the liver region, and set the upper and lower threshold values to 0.43 and 0.56 respectively.

To restrict the seeds from growing outside the liver, we narrowed down the volume. As with the brain tumour, we only let some slices be taken into account, i.e. slices 105 through 167. But, in this case we also cut out a subvolume in the x- and y-direction, constraining the seeds to grow within pixel 29 through 170 with respect to the x-direction, and 79 through 215 for the y-direction. We also set the number of iterations to 35, something that reduced the effect of over-segmentation since the seeds were not allowed to grow for too long. The delta value, which we set to 0.04, also helped reducing this effect.

The algorithm finished in 911 ms, and the resulting images are shown in Figure 6.11. There are some obvious signs of over-segmentation, in spite of the constraints. One problem was that the seeds grew into the tissue that surrounded the ribs. This was inevitable, but the constraint for the x-direction, together with the low number of iterations, reduced this aspect. And of course there are some other small regions and spurious voxels that should not have been part of the region of interest. Still, the liver segmentation performed better than we expected. The various input parameters that we used and the resulting time consume are shown in Table 6.2.

6.2.5 Kidneys

We experimented with segmenting the kidneys in data set A4. They were relatively distinct from the surrounding matter, and were for the most only in direct contact with blood vessels. So it was an appropriate task for our algorithm.

After placing a seed in each of the kidneys, the upper and lower thresholds were set to 0.495 and 1.0 respectively. The algorithm was then executed on the entire volume, and the seeds were allowed to grow for 55 iterations. The total time spent in graphics hardware summed up to 5118 ms.

In this case we did not need to use a delta value nor considering only a subvolume in order to get acceptable results. Table 6.2 summarizes the details from this segmentation, and Figure 6.12 shows the results. The kidneys are nicely segmented with parts of the blood vessels included. Some extra matter around the blood vessels of the kidney to the right is also included, since it has the same level of intensities. There are some holes and gaps within the segmented kidneys, due to that the intensity levels of these voxels are much lower than the rest of the kidneys.

6.3 Comparison with CPU Implementation

The seeded region growing algorithm does not involve operations with high computational intensity, and hence much of the computation time when running the SRG on the GPU will be overhead associated with shader- and render-target (FBO) setup. For each iteration of the **GrowRegion**-shader, a new FBO must be bound and three 2D textures[4] must be specified for the shader, for all image slices beeing processed by the algorithm. Thus, more computations performed per pixel is likely to be more efficient on the GPU. A non-optimal CPU implementation of the SRG algorithm was tested against our GPU based solution. This CPU implementation tries to reflect the GPU method in that all pixels are processed and new seed points are sought in a 6-neighbourhood. This is iterated in order to grow the region. Surprisingly, the GPU implementation was less efficient than the CPU approach. An experiment using data set A1 was carried out on both the GPU and CPU implementations. Same parameters were used in both tests. The GPU/CPU ratio with respect to time consume was approximately 1.13[5]. The same segmentation results were achieved in both approaches. On the other hand, the GPU ran approximately 1.2[6] times faster than the CPU when segmenting the aorta in data set A3. This experiment involved fewer texture slices (140 vs. 260) than the experiment using data set A1, and thus less overhead was required per iteration.

An additional experiment involving more computations was carried out on data set A1. A CPU version of the nonlinear anisotropic diffusion filter was implemented and tested against the GPU implementation. Now the GPU was approximately 6 times faster than the CPU[7]. This supports the claim that more computations performed per fragment is likely to accelerate on the GPU. Much of this advantage stems from the graphics hardware's ability to perform more than one floating-point operation per clock cycle.

[4]Above, current, below.

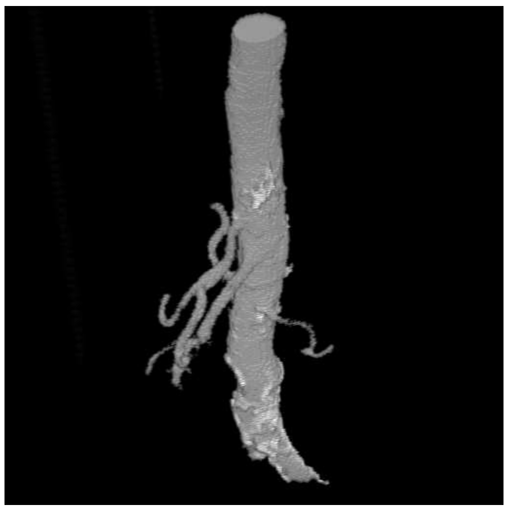
[5]9 seeds, 50 iterations. CPU/GPU time: 8332/9433 ms.

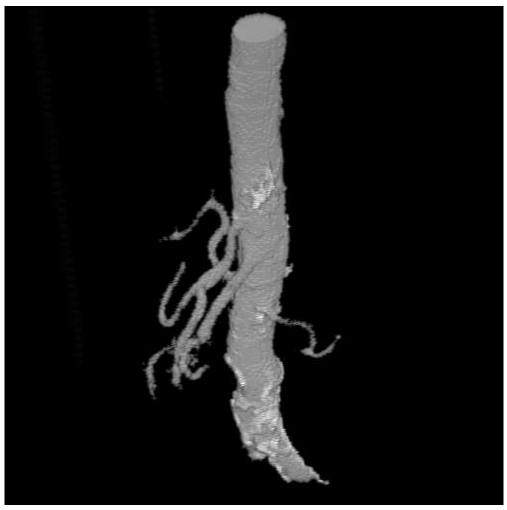
[6]10 seeds, 50 iterations. CPU/GPU time: 4967/4006 ms.

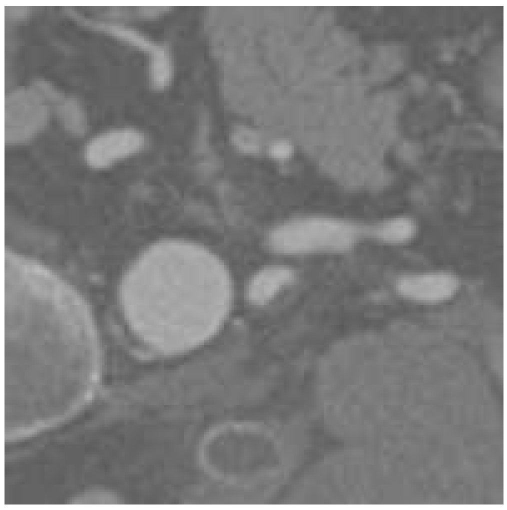
[7]5 steps used. CPU/GPU time: 24366/4046 ms.

6.4 Result Visualizations

In this section, rendered volumes of the segmentation results discussed in Section 6.2 are shown. Each figure consists of the rendered volume and some selected slices to better visualize the quality of the segmentation.


(a) Segmentation of the aorta in data set A1 after 50 iterations.


(b) Segmentation of the aorta in data set A1 using occlusion query, 74 iterations.


(c) Part of image slice from data set A1.

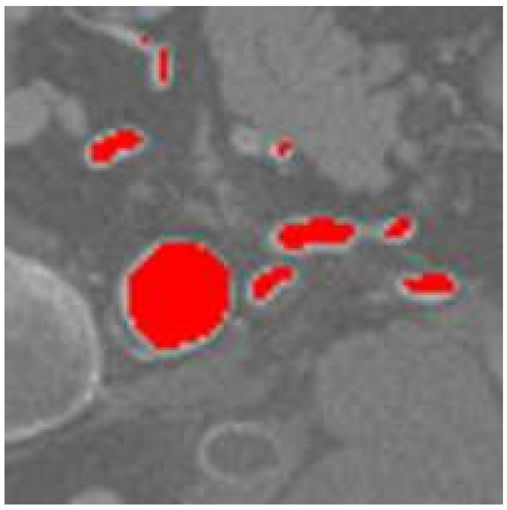
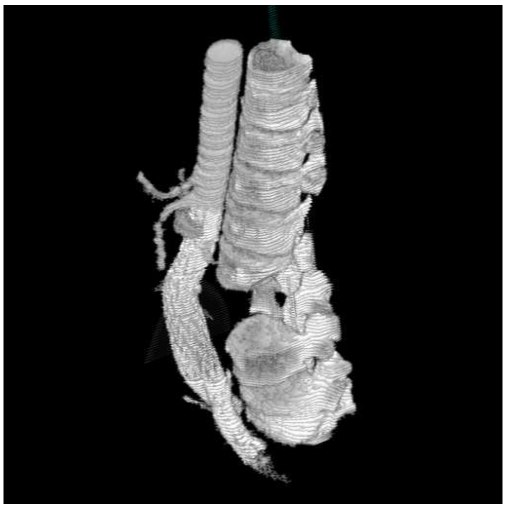
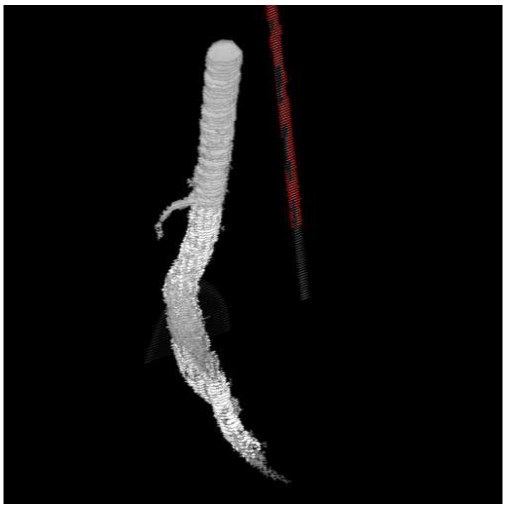
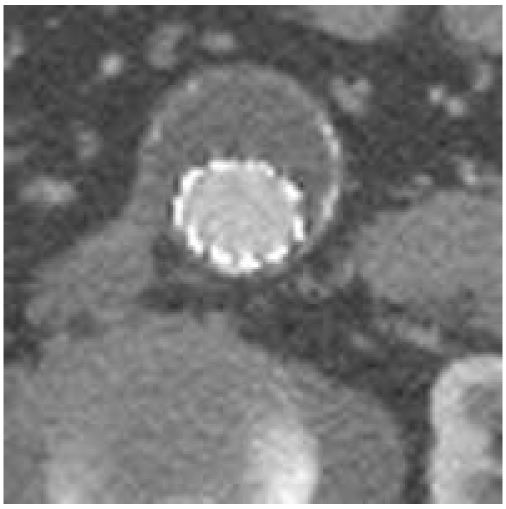

(d) Segmentation of **(c)**.

Figure 6.5: Segmentation of aorta in data set A1.


(a) Segmentation of the aorta in data set A2 after 50 iterations. No delta value used. The segmentation leaks through the weak interface between the aorta and the spinal marrow.


(b) Segmentation of the aorta in data set A2 after 50 iterations. Delta value of 0.05 used to correct over-segmentation.


(c) Part of image slice from data set A2.

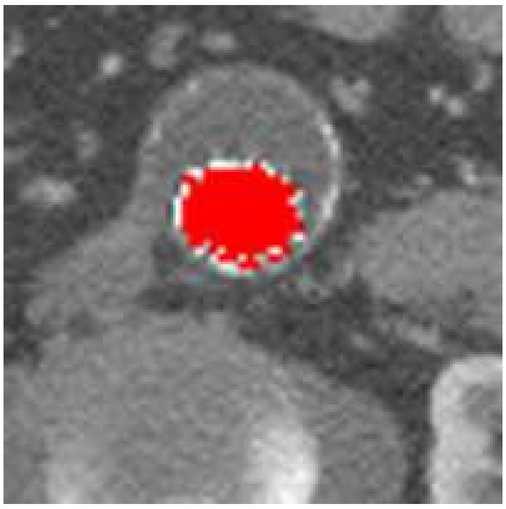
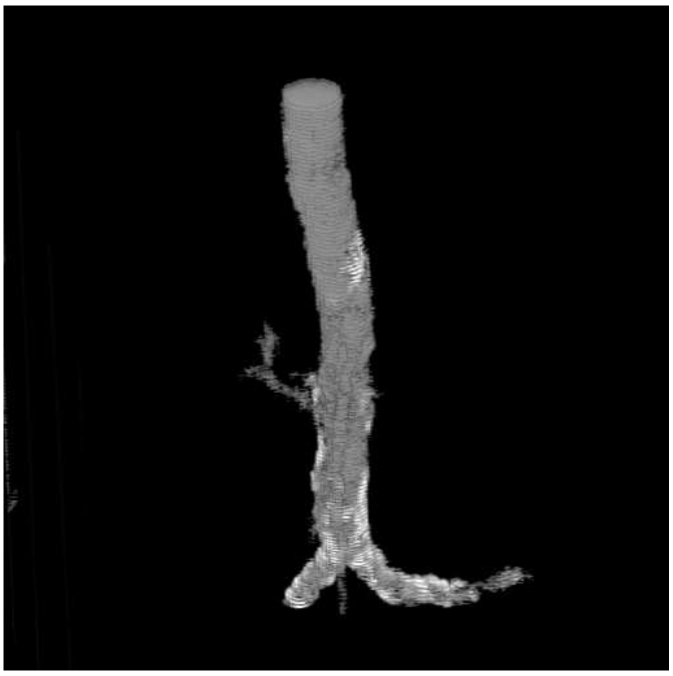
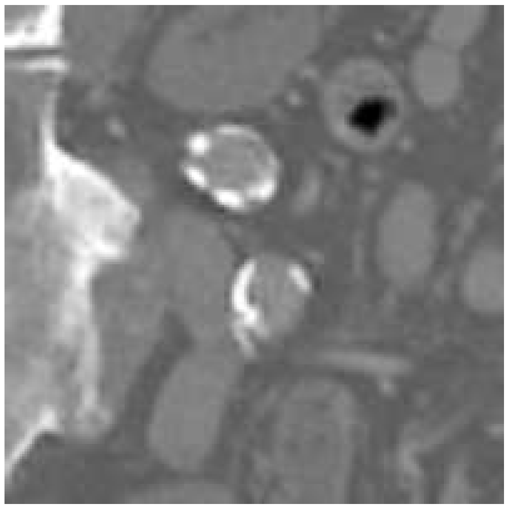

(d) Segmentation of **(c)**.

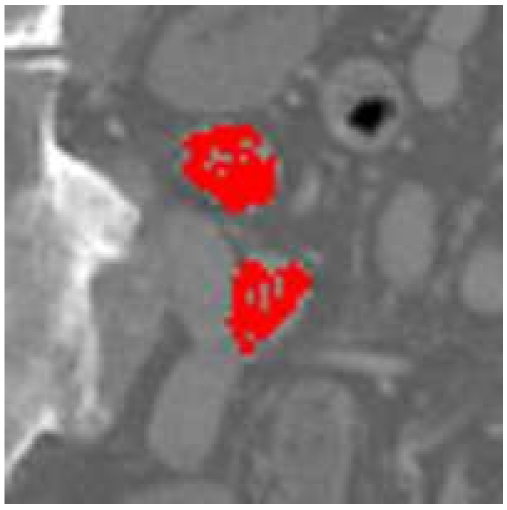
Figure 6.6: Segmentation of aorta in data set A2.



(a) Segmentation of the aorta in data set A3 after 50 iterations.

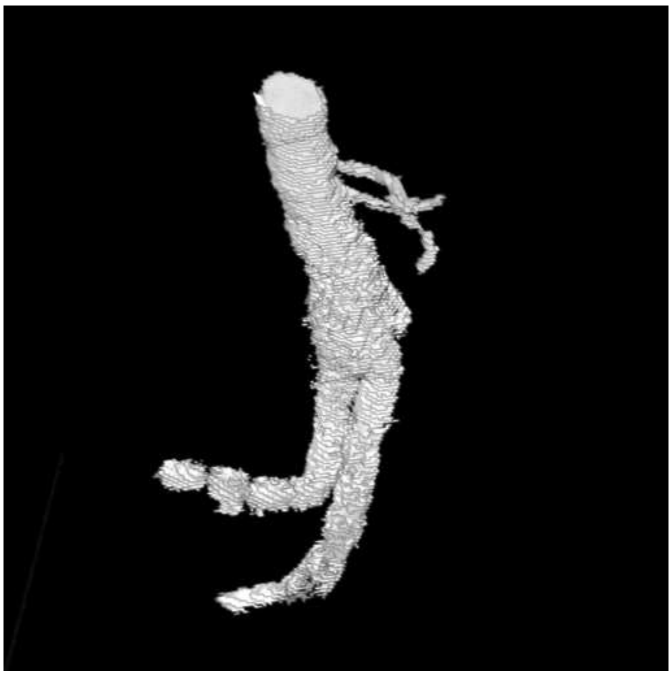


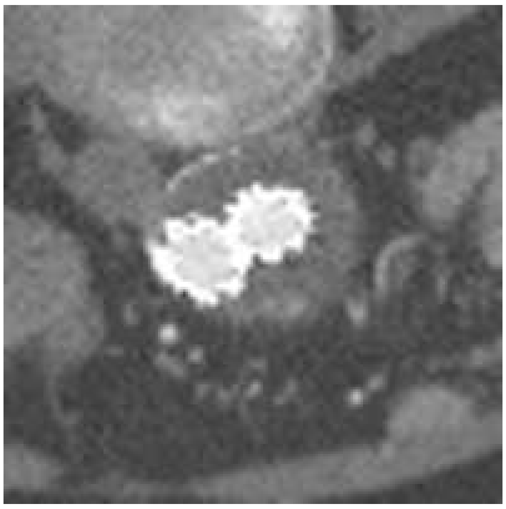
(b) Part of image slice from data set A3.



(c) Segmentation of **(b)**.

Figure 6.7: Segmentation of aorta in data set A3.


(a) Segmentation of the aorta in data set A4 after 50 iterations.


(b) Part of image slice from data set A4.

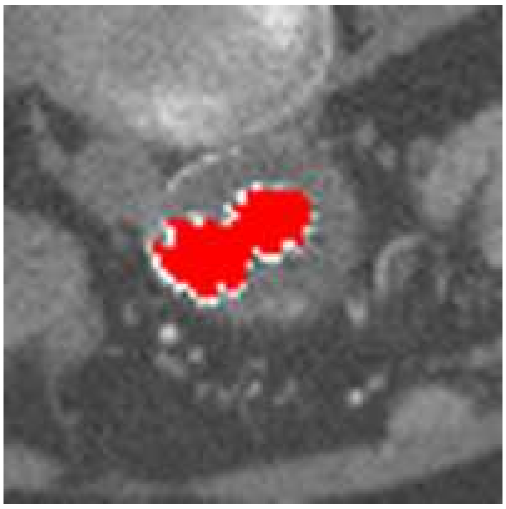
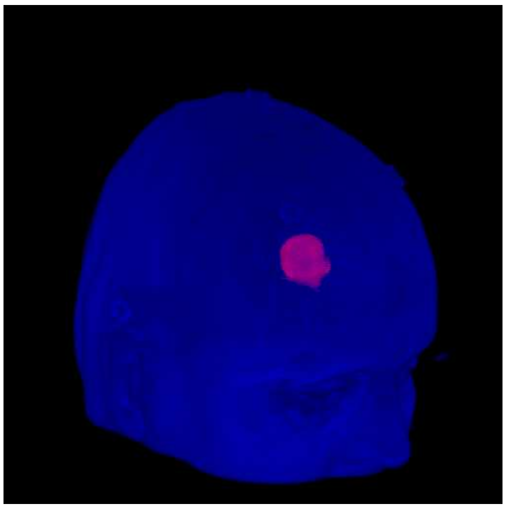
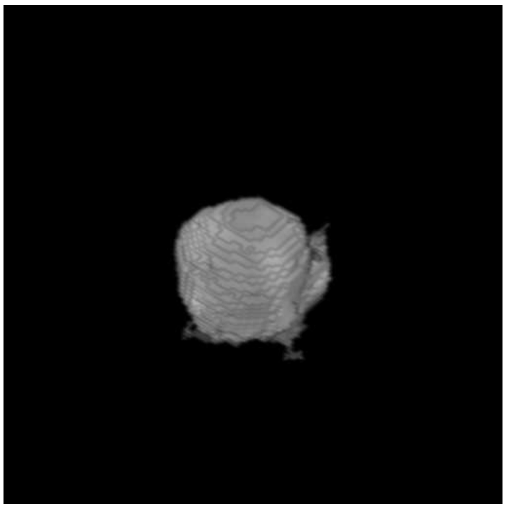
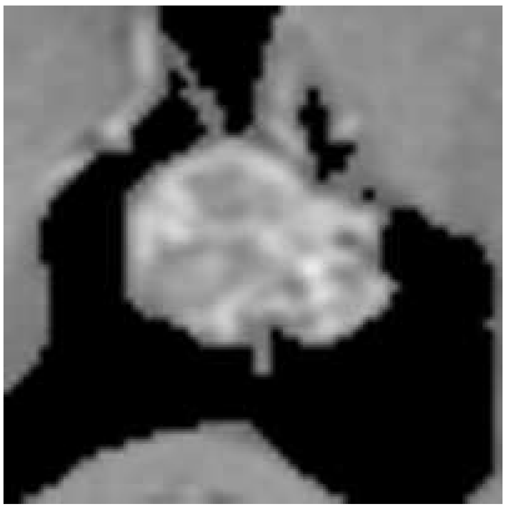

(c) Segmentation of **(b)**.

Figure 6.8: Segmentation of aorta in data set A4.


(a) Segmentation of the brain tumour in data set H1 after 30 iterations. Complete volume rendered.


(b) The brain tumour isolated.


(c) Part of image slice from data set H1.

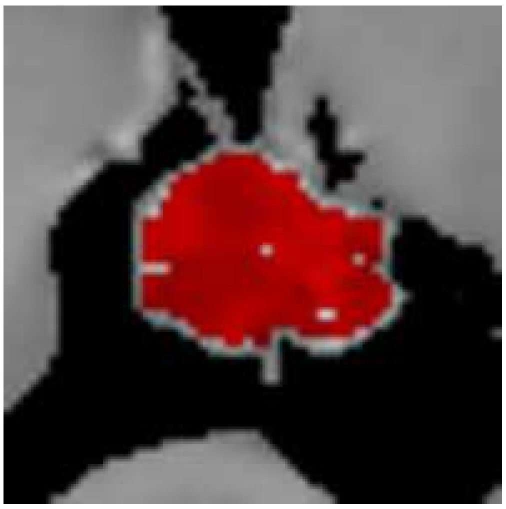
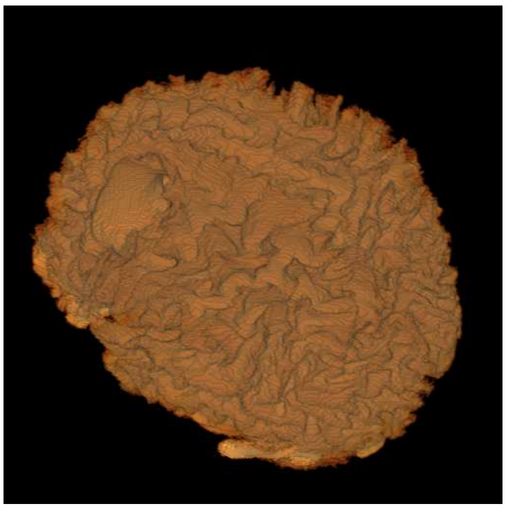
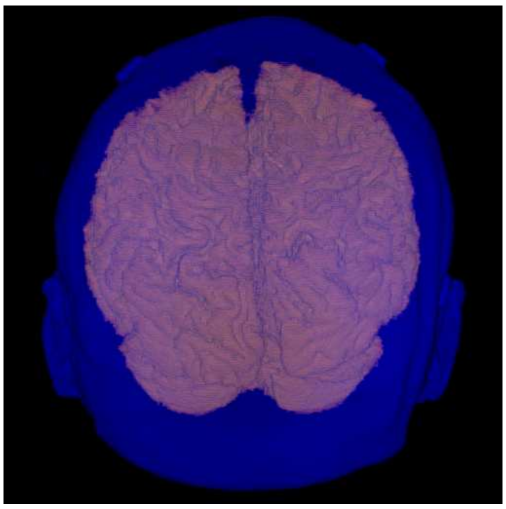
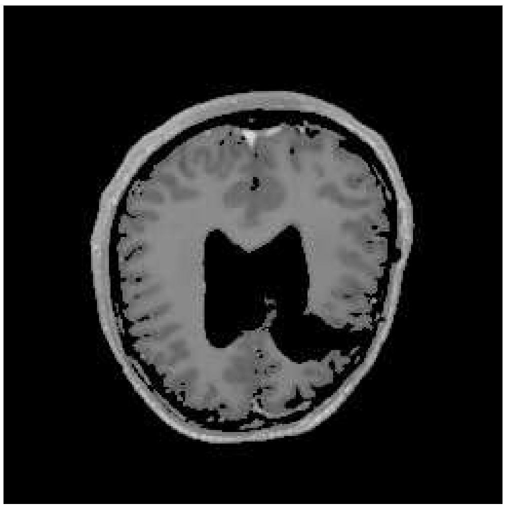

(d) Segmentation of **(c)**.

Figure 6.9: Segmentation of brain tumour in data set H1.


(a) Segmentation of the cerebral cortex in data set H1 after 80 iterations.


(b) Segmentation of the cerebral cortex in data set H1. Complete volume rendered.


(c) Part of image slice from data set H1.

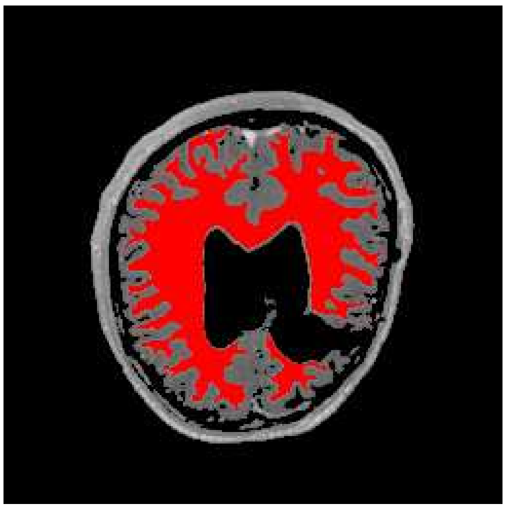
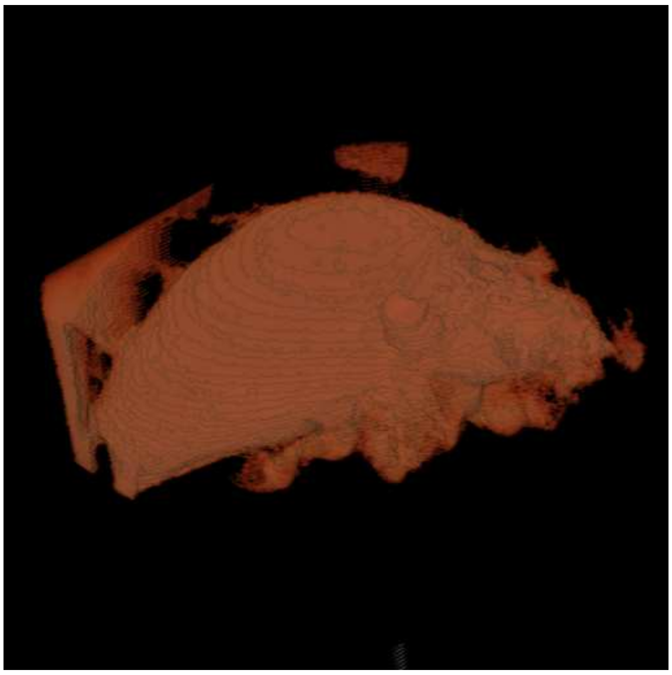
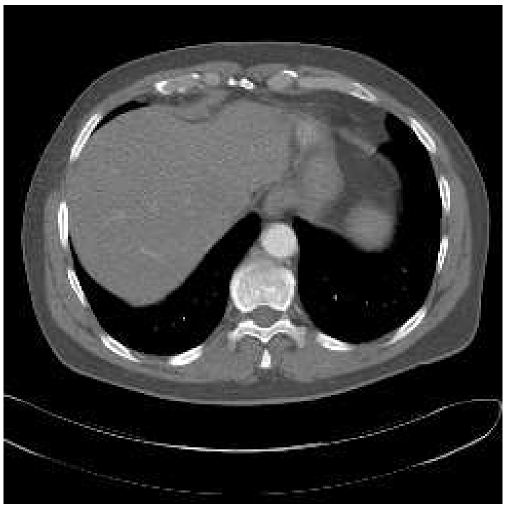

(d) Segmentation of **(c)**.

Figure 6.10: Segmentation of cerebral cortex in data set H1.


(a) Segmentation of the liver in data set A2 after 35 iterations.


(b) Part of image slice from data set A2.

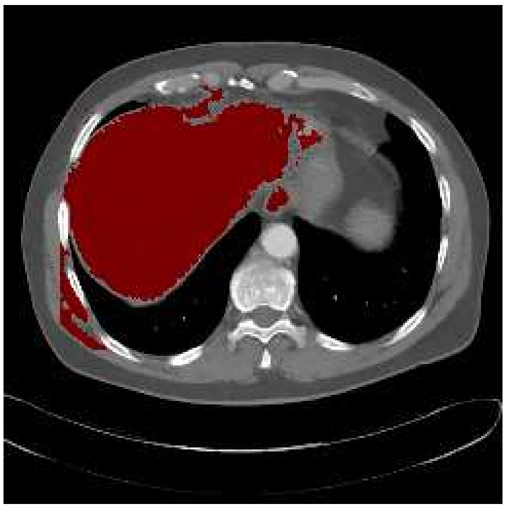
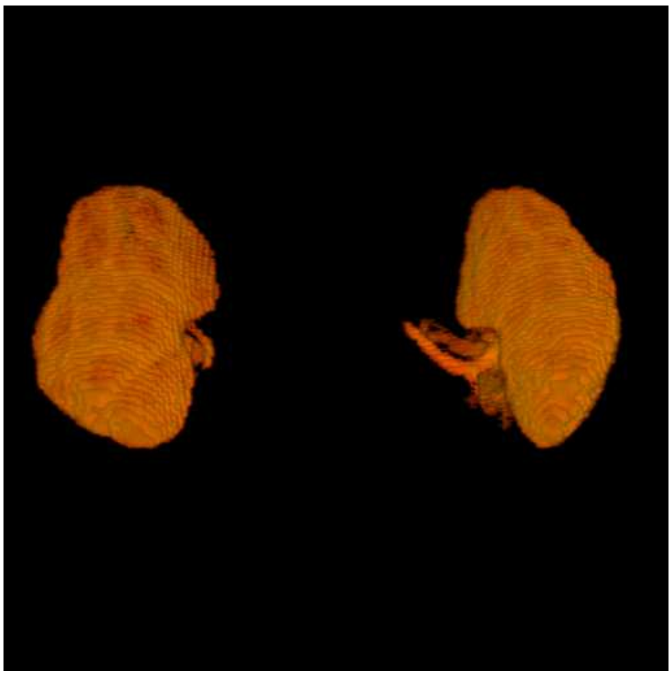
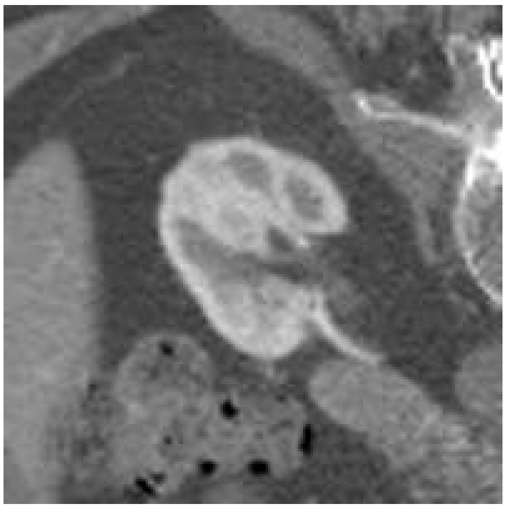

(c) Segmentation of **(b)**.

Figure 6.11: Segmentation of liver in data set A2.


(a) Segmentation of the kidneys in data set A4 after 55 iterations.


(b) Part of image slice from data set A4.

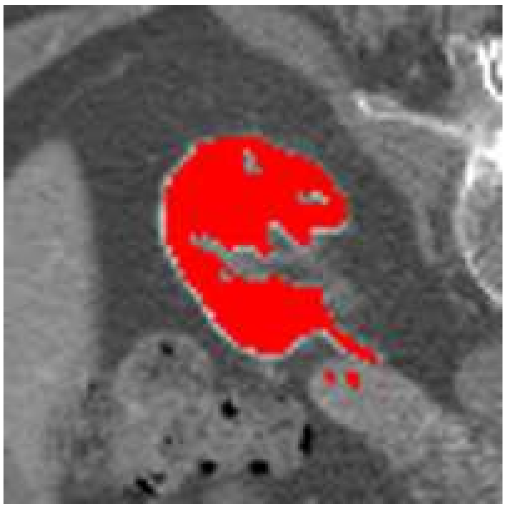

(c) Segmentation of **(b)**.

Figure 6.12: Segmentation of kidneys in data set A4.

Chapter 7

Conclusions

In this Thesis, we have looked at the possibility of using the graphics processing unit for medical image segmentation. By using state of the art technology, e.g. the framebuffer object extension, seeded region growing (SRG) has been successfully implemented on commodity graphics hardware, along with two pre-processing filters, i.e. median filter and nonlinear anisotropic diffusion filter. In addition, we created a volume visualizer. Overall, however, our main attention has merely been to the segmentation task.

7.1 Discussion

Nowadays, the GPUs are fully programmable parallel processors that in some cases can outperform the CPU with over an order of magnitude. Their big slopes with respect to performance, measured in GFLOPS, and the high internal bandwidth with a memory system that can be addressed very fast, make them appealing to programmers. In addition, the GPU relieves the CPU, so it can be left with other tasks. If the results are to be visualized, data transfer from CPU to GPU is avoided, since the data already resides on the graphics card.

General-purpose GPU programming requires an effort in mapping a problem within the hardware constraints of the GPU. Special care must be taken, since the basic way of CPU programming is altered. All operations performed on a fragment must be general so that it can be performed on every fragment, i.e. there can be no indivdual per-fragment operations, and a write operation is only allowed for the record in process. The fact that there are no global registers makes the task even harder for some problems, e.g. averaging and counting.

Despite this, we found a solution for mapping the seeded region growing algorithm onto the GPU, hoping to fully utilize its parallelism and enormous computational power. Our GPU implementation consists of four shaders that are consecutively executed to grow reigons based on user-defined seed points, threshold interval and connectivity criteria. If required, a delta value may be used to prevent the segmentation to leak through weak interfaces.

The implementation gave the anticipated segmentation results. SRG is a simple algorithm that performs well when the voxels within the region of interest have similar luminance values that also are releative different from the surrounding voxels. There are also low computational costs associated with using it. Despite this, we have succeeded in segmenting structures from different medical volume data sets.

When looking at the time consume, CPU implementations perform better. An optimized, recursive CPU implementation only processes the smallest number of necessary voxels. Basically, the GPU must process all voxels. There is no simple way for the GPU application to specify which voxels to process. We have developed a CPU reference implementation of the SRG algorithm. This non-optimal implementation tries to reflect the GPU based solution. The GPU version was less efficient than this CPU implementation, due to the overhead associated with shader- and render-target (FBO) setup, and the fact that the SRG algorithm has low computational costs and performs few per-fragment operations. The SRG implementation was not able to fully utilize the GPU's computational power. More complex tasks involving higher computational intensity like the anisotripic diffusion filter, performed better on the GPU. In this case, more floating-point operations were performed per fragment, and the graphics card's computational capacity and parallelism were more utilized.

The current NVIDIA display driver does not yet support render-to-3D-texture using the framebuffer object extension (FBO). Utilizing 3D textures, the overhead associated with render setup may decrease and result in a performace gain. Also, the fact that C evaluates boolean expression in a short-circuiting manner, can reduce the computation time. This is not supported by Cg.

7.2 Final Conclusions

We have ported the seeded region growing algorithm from the CPU programming model to the GPU programming model, and thus implemented an iterative version of SRG on the GPU, hoping for a speed-up in time consume. The segmentation results were as expected, restricted by the constraints of the algorithm. Anatomical structures like the aorta, a brain tumour, the cerebral cortex, the kidneys and the liver have been successfully segmented by our GPU based SRG application. However, the CPU implementation performed slightly better than the GPU implementation when measuring the computation time. But, reducing the number of image slices (textures), less overhead was required per iteration of the region growing process, and the GPU version required less computation time than the CPU. A volume renderer was also implemented and incorporated in our graphics application. This was used to visualize the volumes before and after segmentation, and contributed to an easy inspection of the segmentation results. We used a simple and easy-to-implement method based on 2D textures and blending in OpenGL.

When designing the SRG shaders, much effort has been on optimizations. Having in mind the amount of data being processed when dealing with volume data, optimized algorithms are clearly required to achieve acceptable results. We have succeeded in implementing an optimized region growing algorithm which does not include any conditionals, nor loops. Also, the alpha-test has been used as a computation mask reducing the amount of data being processed. Weeding out fragments by initially executing an exclude-shader in one render pass, resulted in a performace gain, and the **GrowRegion**-shader only had to consider potential seed points. The GPU is more utilized when algorithms can be structured as *streaming* computations, meaning that a kernel is independently evaluated on each input value. The performance gain is even more evident when more computations are performed per fragment. This was proved when comparing a CPU and GPU implementation of the compute-intensive anisotropic diffusion filter. For this task, the GPU outperformed the CPU by a factor of 6. Thus, more sophisticated segmentation methods like level-sets and active contours or surfaces are likely to accelerate on GPUs.

Chapter 8

Future Work

From our literature study in [2], we moved on to implement seeded region growing on the GPU. We have also implemented two pre-processing filters, i.e. median filter and nonlinear anisotropic diffusion filter, in addition to a volume renderer.

8.1 Further Implementations

An interesting approach for further implementations is to consider more complex segmentation methods, to fully exploit the computational resources of graphics hardware. In relation to this, we find active contour or surface models, and level-set based methods particularly appealing. The latter is based on the numeric of weak solutions to surface propagation and involves computations for solving a partial differential equation on a volume. This can surely benefit from the parallelism and computational power offered by GPUs.

The basic SRG algorithm could be extended to include more complex merging criteria, e.g. gradient values and other global or local image characteristics. In addition, the merging criteria could be expressed as a diffusion process.

As discussed in Chapter 5, our volume renderer only exploits 2D textures, and some viewing artifacts exist as a consequence of this. Utilizing 3D textures for volume visualization would be an improvement. An animation of the segmentation evolution is also an interesting future challenge. This would allow a user to observe the segmentation in real time, and cancel the running if the results are not acceptable. This should not affect the running time of the segmentation, and should potentially lead to fully utilization of the GPU's computation units.

8.2 Integration into CustusX

Since the CPU based version of seeded region growing turned out to be faster than the GPU implementation in some cases, a direct integration into CustusX seems somehow useless. However, a GPU framework that incorporates different filtering and more complex segmentation methods, may serve as a module within CustusX. It may also turn out to be possible to use the GPU for other medical tasks that need a high amount of processing power, e.g. volume to volume registration.

Bibliography

[1] R. Adams and L. Bischof. Seeded Region Growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:641–647, 1994.

[2] M. Botnen and H. Ueland. The GPU as a Computational Resource in Medical Image Processing, 2004.

[3] I. Buck. Brook Spec v0.2. http://merrimac.stanford.edu/brook/, 2003.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. BrookGPU. http://graphics.stanford.edu/projects/brookgpu.

[5] J. E. Cates, Aa. E. Lefohn, and R. T. Whitaker. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis*, 8:217–231, 2004.

[6] NVIDIA Corporation. Cg Toolkit. http://developer.nvidia.com/Cg, 2004.

[7] NVIDIA Corporation. Technical Brief: The GeForce 6 Series of GPUs, 2004.

[8] NVIDIA Corporation. NVIDIA GPU Programming Guide, 2005.

[9] M. McCool et al. Sh: A high-level metaprogramming language for modern GPUs. http://libsh.org, 2003.

[10] R. Fernando, M. Harris, M. Wloka, and C. Zeller. Programming Graphics Hardware. NVIDIA Corporation, 2004.

[11] G. Gerig, R. Kikinis, O. Kübler, and F.A. Jolesz. Nonlinear Anisotropic Filtering of MRI Data. *IEEE Transactions on Medical Imaging*, 11(2):221–232, June 1992.

[12] S. Green. The OpenGL Framebuffer Object Extension. NVIDIA Corporation, 2005.

[13] M. Hadwiger, C. Berger, and H. Hauser. High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware. *IEEE Visualization*, 2003.

[14] Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller. Hardware-accelerated high-quality reconstruction on pc hardware. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001*, pages 105–112. Aka GmbH, 2001.

[15] Markus Hadwiger, Ivan Viola, Thomas Theußl, and Helwig Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Vision, Modeling and Visualization 2002*, pages 155–162, nov 2002.

[16] M. Harris. GPGPU: General-Purpose Computation on GPUs. EG 2004, 2004.

[17] M. Hopf and T. Ertl. Accelerating 3D Convolution using Graphics Hardware. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*. IEEE Computer Society, 1999.

[18] M. Hopf and T. Ertl. Hardware-Based Wavelet Transformations. In *Workshop of Vision, Modelling, and Visualization (VMV '99)*. infix, 1999.

[19] M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*. infix, 2000.

[20] M. Hopf and T. Ertl. Hardware Accelerated Wavelet Transformations. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, 2000.

[21] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models. *International Journal of Computer Vision*, pages 321–331, 1988.

[22] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language. 3Dlabs, Inc. Ltd, 2004.

[23] T. McInerney. Topology Adaptive Deformable Surfaces for Medical Image Volume Segmentation. *IEEE Transactions on medical imaging*, 18, 1999.

[24] T. C. Pedersen. Segmenting Medical Images on the GPU. University of Aarhus, 2005.

[25] M. Rumpf and R. Strzodka. Level Set Segmentation in Graphics Hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, volume 3, pages 1103–1106, 2001.

[26] A. Sherbondy, M. Houston, and S. Napel. Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware. *IEEE Visualization*, 2003.

[27] A. Shetty, A. Nitin, Ashwin K., Pramod C., and Venkatesh N. GPU based Volume Rendering of Medical Images. Philips Medical Systems, Philips Innovation Campus, Bangalore.

[28] SINTEF. Navigasjonssystemet CustusX. http://www.sintef.no, 2005.

[29] Jayaram K. Udupa, Vicki R. LaBlanc, Hilary Schmidt, Celina Imielinska, Punam K. Saha, George J. Grevera, Ying Zhuge, L. M. Currie, Pat Molholt, and Yinpeng Jin. Methodology for Evaluating Image Segmentation Algorithms. volume 4684, pages 266–277. SPIE, 2002.

[30] Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller. Hardware-based nonlinear filtering and segmentation using high-level shading languages. In *Proceedings of IEEE Visualization 2003*, pages 309–316. G. Turk, J. van Wijk, K. Moorhead, oct 2003.

[31] C. Wynn. Using P-Buffers for Off-Screen Rendering in OpenGL. NVIDIA Corporation.

[32] R. Yang and G. Welch. Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware. Dept. of Computer Science, University of North Carolina at Chapel Hill.

Appendix A

Seeded Region Growing - Cg Shaders

A.1 Exclude

```
void exclude(
             half2 texCoord : TEXCOORD0,
             uniform half uThreshold,
             uniform half lThreshold,
             out half4 color : COLOR,
             uniform samplerRECT texture)
{
    color = texRECT(texture, texCoord);

    half f = step(lThreshold, color.b) * step(color.b, uThreshold);

    color.ra = half2(1, f);
}
```

A.2 Set Seed

```
void setSeed(
             half2 texCoord : TEXCOORD0,
             uniform half2 seedPoint,
             out half4 color : COLOR,
             uniform samplerRECT texture)
{
    color = texRECT(texture, texCoord);
    half2 lower = seedPoint + half2(-1, -1);
    half2 upper = seedPoint + half2(1, 1);

    if (texCoord.x > lower.x && texCoord.x < upper.x &&
        texCoord.y > lower.y && texCoord.y < upper.y)
    {
        color.r = 0;
    }

}
```

A.3 Grow Region

A.3.1 Grow Region 3D

```
void growRegion3DNormal(
                  half2 texCoord : TEXCOORD0,
                  out half4 color : COLOR,
                  uniform samplerRECT up,
                  uniform samplerRECT current,
                  uniform samplerRECT down)
{
    color = texRECT(current, texCoord);

    if (color.r == 0)
        discard;

    color.r = texRECT(current, texCoord + half2(-1, 0)).r *
        texRECT(current, texCoord + half2(1, 0)).r *
        texRECT(current, texCoord + half2(0, -1)).r *
        texRECT(current, texCoord + half2(0, 1)).r *
        texRECT(up, texCoord).r *
        texRECT(down, texCoord).r;
}
```

A.3.2 Grow Region 3D Using a Delta Value

```
void growRegion3D(
                  half2 texCoord : TEXCOORD0,
                  out half4 color : COLOR,
                  uniform half delta,
                  uniform samplerRECT up,
                  uniform samplerRECT current,
                  uniform samplerRECT down)
{
    color = texRECT(current, texCoord);

    if (color.r == 0)
        discard;

    half4 w = texRECT(current, texCoord + half2(-1, 0));
    half4 e = texRECT(current, texCoord + half2(1, 0));
    half4 n = texRECT(current, texCoord + half2(0, -1));
    half4 s = texRECT(current, texCoord + half2(0, 1));
    half4 u = texRECT(up, texCoord);
    half4 d = texRECT(down, texCoord);

    half r = (w.r + step(delta, abs(color.b-w.b))) *
            (e.r + step(delta, abs(color.b-e.b))) *
            (n.r + step(delta, abs(color.b-n.b))) *
            (s.r + step(delta, abs(color.b-n.b))) *
            (u.r + step(delta, abs(color.b-u.b))) *
            (d.r + step(delta, abs(color.b-d.b)));

    color.r = r;
}
```

A.3.3 Grow Region 2D

```
void growRegion(
                half2 texCoord : TEXCOORD0,
                out half4 color : COLOR,
                uniform samplerRECT texture)
{
    color = texRECT(texture, texCoord);

    if (color.r == 0)
        discard;

    half r = texRECT(texture, texCoord + half2(-1, 0)).r *
                texRECT(texture, texCoord + half2(0, -1)).r *
                texRECT(texture, texCoord + half2(1, 0)).r *
                texRECT(texture, texCoord + half2(0, 1)).r;

    color.r = r;
}
```

A.4 Shade Segmented

```
void shadeSegmented(
                    half2 texCoord,
                    out half4 color : COLOR,
                    uniform half keepUnsegmented,
                    uniform samplerRECT texture)
{
    color = texRECT(texture, texCoord);

    color.rgb = half3(color.b*(1-color.r),
        color.b*(1-color.r)*(1-keepUnsegmented),
        color.b*color.r*keepUnsegmented +
        color.b*(1-color.r)*(1-keepUnsegmented));
}
```

Appendix B

Filtering - Cg Shaders

B.1 Median filter

```
void medianH(
          in half4 texCoord : TEXCOORD0,
          out half4 color : COLOR,
          uniform samplerRECT texture)
{
    half a = texRECT(texture, texCoord + half2(-1, 0)).r;
    half b = texRECT(texture, texCoord + half2(0, 0)).r;
    half c = texRECT(texture, texCoord + half2(1, 0)).r;

    half med = max(min(a, b), min(max(a, b), c));

    color = half4(med, med, med, 1);
}

void medianV(
          in half4 texCoord : TEXCOORD0,
          out half4 color : COLOR,
          uniform samplerRECT texture)
{

    half a = texRECT(texture, texCoord + half2(0, -1)).r;
    half b = texRECT(texture, texCoord + half2(0, 0)).r;
    half c = texRECT(texture, texCoord + half2(0, 1)).r;

    half med = max(min(a, b), min(max(a, b), c));

    color = half4(med, med, med, 1);
}
```

B.2 Anisotropic Diffusion Filter

B.2.1 2D Implmentation

```
float grad(samplerRECT decal, float2 texCoord)
{
    float hx = 0;
    hx += texRECT(decal, texCoord + float2(-1, -1));
    hx -= texRECT(decal, texCoord + float2(1, -1));
    hx += 2.0f*texRECT(decal, texCoord + float2(-1, 0));
    hx -= 2.0f*texRECT(decal, texCoord + float2(1, 0));
    hx += texRECT(decal, texCoord + float2(-1, 1));
    hx -= texRECT(decal, texCoord + float2(1, 1));
    hx /= 4.0f;

    float hy = 0;
    hy += texRECT(decal, texCoord + float2(-1, -1));
    hy += 2.0f*texRECT(decal, texCoord + float2(0, -1));
    hy += texRECT(decal, texCoord + float2(1, -1));
    hy -= texRECT(decal, texCoord + float2(-1, 1));
    hy -= 2.0f*texRECT(decal, texCoord + float2(0, 1));
    hy -= texRECT(decal, texCoord + float2(1, 1));
    hy /= 4.0f;

    return abs(hx) + abs(hy);
}


float c(float gradient, float k)
{
    float innerProd = (gradient/k)*(gradient/k);
    return exp(-innerProd);
}

float dI(samplerRECT decal, float2 texCoord, float k)
{
    float gradient = grad(decal, texCoord);
    float c_ = c(gradient, k);
    float e = c_*(texRECT(decal, texCoord + float2(1, 0)) -
        texRECT(decal, texCoord + float2(0, 0)));
    float w = c_*(texRECT(decal, texCoord + float2(0, 0)) -
        texRECT(decal, texCoord + float2(-1, 0)));
    float n = c_*(texRECT(decal, texCoord + float2(0, -1)) -
        texRECT(decal, texCoord + float2(0, 0)));
    float s = c_*(texRECT(decal, texCoord + float2(0, 0)) -
        texRECT(decal, texCoord + float2(0, 1)));

    return (e - w + n - s);
}

void main(
          float2 texCoord : TEXCOORD0,
          out float4 color : COLOR,
          uniform float2 k_step,
          uniform samplerRECT texture)
{
    float col = texRECT(texture, texCoord);
    float gray = col + k_step.y*dI(texture, texCoord, k_step.x);
    color = float4(gray, gray, gray, 1);
}
```

B.2.2 3D Implementation

```
float grad(samplerRECT lower,
    samplerRECT current,
    samplerRECT upper,
    float2 texCoord)
{
    float hx = 0;
    hx += texRECT(current, texCoord + float2(-1,-1));
    hx += texRECT(current, texCoord + float2(-1, 1));
    hx += 4.0f * texRECT(current, texCoord + float2(-1, 0));
    hx += texRECT(upper, texCoord + float2(-1, 0));
    hx += texRECT(lower, texCoord + float2(-1, 0));

    hx -= texRECT(current, texCoord + float2(1, 1));
    hx -= texRECT(current, texCoord + float2(1,-1));
    hx -= texRECT(upper, texCoord + float2(1, 0));
    hx -= texRECT(lower, texCoord + float2(1, 0));
    hx -= 4.0f * texRECT(current, texCoord + float2(1, 0));

    hx /= 8.0f;

    float hy = 0;
    hy += texRECT(current, texCoord + float2(-1, -1));
    hy += texRECT(current, texCoord + float2(1, -1));
    hy += 4.0f * texRECT(current, texCoord + float2(0, -1));
    hy += texRECT(upper, texCoord + float2(0, -1));
    hy += texRECT(lower, texCoord + float2(0, -1));

    hy -= texRECT(current, texCoord + float2(1, 1));
    hy -= texRECT(current, texCoord + float2(-1, 1));
    hy -= 4.0f * texRECT(current, texCoord + float2(0, 1));
    hy -= texRECT(upper, texCoord + float2(0, 1));
    hy -= texRECT(lower, texCoord + float2(0, 1));

    hy /= 8.0f;
    float hz = 0;
    hz += texRECT(upper, texCoord + float2(-1, 0));
    hz += texRECT(upper, texCoord + float2(0, -1));
    hz += texRECT(upper, texCoord + float2(1, 0));
    hz += texRECT(upper, texCoord + float2(0, 1));
    hz += 4.0f*texRECT(upper, texCoord);

    hz -= texRECT(lower, texCoord + float2(-1, 0));
    hz -= texRECT(lower, texCoord + float2(0, -1));
    hz -= texRECT(lower, texCoord + float2(1, 0));
    hz -= texRECT(lower, texCoord + float2(0, 1));
    hz -= 4.0f*texRECT(lower, texCoord);

    hz /= 8.0f;

    return abs(hx) + abs(hy) + abs(hz);
}

float c(float gradient, float k)
{
    float innerProd = (gradient/k)*(gradient/k);
    return exp(-innerProd);
}

float dI(samplerRECT lower,
    samplerRECT current,
    samplerRECT upper,
    float2 texCoord,
    float k)
{
    float gradient = grad(lower, current, upper, texCoord);
    float c_ = c(gradient, k);
    float e = c_*(texRECT(current, texCoord + float2(1, 0)) -
        texRECT(current, texCoord));
    float w = c_*(texRECT(current, texCoord) -
        texRECT(current, texCoord + float2(-1, 0)));
    float n = c_*(texRECT(current, texCoord + float2(0, 1)) -
        texRECT(current, texCoord));
    float s = c_*(texRECT(current, texCoord) -
        texRECT(current, texCoord + float2(0, -1)));
    float up = c_*(texRECT(upper, texCoord) -
        texRECT(current, texCoord));
    float down = c_*(texRECT(current, texCoord) -
        texRECT(lower, texCoord));

    return (e - w + n - s + up - down);
}

void main(
          float2 texCoord : TEXCOORD0,
          out float4 color : COLOR,
          uniform float2 k_step,
          uniform samplerRECT lower,
          uniform samplerRECT current,
          uniform samplerRECT upper)
{
    float col = texRECT(current, texCoord);
    float gray = col + k_step.y *
        dI(lower, current, upper, texCoord, k_step.x);
    color = float4(gray, gray, gray, 1);
}
```

B.3 Threshold

```
void main(
          float2 texCoord : TEXCOORD0,
          uniform float threshold,
          out float4 color : COLOR,
          uniform samplerRECT texture)
```

```
{
    color = texRECT(texture, texCoord);
    color.rgb = color.rgb * step(threshold, color.r);
}
```