GPU-Based Volume Segmentation

Stefan Schenke[1], Burkhard C. Wünsche[2] and Joachim Denzler[1]

[1]Friedrich-Schiller-Universität Jena, Lehrstuhl für Bildverarbeitung, D-07740 Jena, Germany.

[2]University of Auckland, Dept. of Computer Science, Private Bag 92019, Auckland, New Zealand.

Email: burkhard@cs.auckland.ac.nz

Abstract

Volume segmentation is an important part of any medical image analysis framework used for diagnoses, treatment planning and biomedical modelling and visualisation. Recent advances in modern graphics hardware have made it possible to perform general purpose computing on the GPU. In this paper we survey and analyse the current state-of-the-art of GPU-based volume segmentation algorithms. Limitations of GPU-based implementations due to the architecture of graphics hardware are discussed and guidelines for developing new GPU-based algorithms are given. We conclude with the GPU-based implementation of a hybrid volume segmentation method which makes use of the latest GPU features such as frame buffer objects. We present a framework for the interactive control and visualisation of the segmentation process and we demonstrate the speed advantage achieved by using the GPU.

Keywords: volume segmentation, GPU programming, GPGPU

1 Introduction

Image segmentation is the process of partitioning an image into non-intersecting regions such that each region is homogeneous with respect to some pre-defined similarity criterion. Applications of image segmentation range from computer vision to medical image analysis which usually involves 3D image data sets (volume segmentation).

In order to obtain good segmentation results it is often necessary to enhance image quality using low-level pre-processing steps to correct shading, reduce noise and suppress other system dependent influences. The quality of the segmentation output is crucial for the success of subsequent high-level image analysis and classification procedures. Segmentation results are affected by issues such as spatial resolution, poor contrast, ill-defined boundaries, noise and other acquisition artifacts. Consequently many researchers believe that image data alone is not sufficient and instead user guidance or a priori knowledge, such as the expected shape or spatial position of an object, is necessary [1].

Over the past decades numerous segmentation methods have been developed but none performs consistently well for all applications. Instead the best methods are usually highly specialised for specific image classes or problems, especially when they integrate a priori knowledge about features like shape or position of the objects to segment. Suri et al. point to more than 1000 publications of image segmentation methods in the medical field alone [2].

It is possible to broadly categorise image segmentation methods into three classes:

- Pixel-based methods directly classify each individual pixel based on its feature value utilising histograms or other global measures.
- Edge-based methods use edge information and minimum cost functions to determine the boundaries of homogeneous regions.
- Region-based methods build object boundaries using a pre-defined similarity criterion which pixels within a region have to match.

A survey and review of image segmentation techniques can be found in [3].

In this paper we investigate volume segmentation algorithms specifically developed for modern graphics hardware. The parallel computational resources of commodity graphics hardware and recent improvements of their programming flexibility have made them a well suited platform for image processing tasks, including image segmentation. We survey and analyse existing algorithms, discuss limitations imposed by the GPU architecture and we present and evaluate our own implementation of an interactive GPU-based hybrid volume segmentation algorithm.

2 General-Purpose Computation on GPUs (GPGPU)

The latest graphics chips provide enormous memory bandwidth and computational power which are key prerequisites for efficiently solving many scientific problems. The NVIDIA GeForce 6 Ultra GPU, for instance, reaches sustained memory transfer rates of up to 35.2 GByte per second and the ATI Radeon X800XT achieves a sustained computational power of 63 GFLOPS, compared to the theoretical peak of 14.8 GFLOPS on a 3.7GHz Intel Pentium 4 SSE2 CPU with 6.4GB per second main memory access [4, 5].

2.1 Programmable Graphics Hardware

Graphics chips incorporate nowadays powerful and flexible architectures providing fully programmable processing units with support for vectorised 32-bit IEEE floating-point operations. High-level programming languages have been developed to easily exploit the flexibility of the vertex and pixel pipelines. The latest improvements of current GPUs comprise vertex texture access, full MIMD branching support in vertex processors as well as limited branching capabilities in the fragment pipeline.

Despite the computational speed, increased precision and rapidly extending programmability, current graphics hardware lacks some fundamental computing constructs: integer arithmetic and associated bit-wise shift and logical operations, like AND, OR, XOR, NOT are not supported on graphics hardware. The architecture of current high-end GPUs reflects the flow of data (*graphics pipeline*) and is illustrated in figure 1.

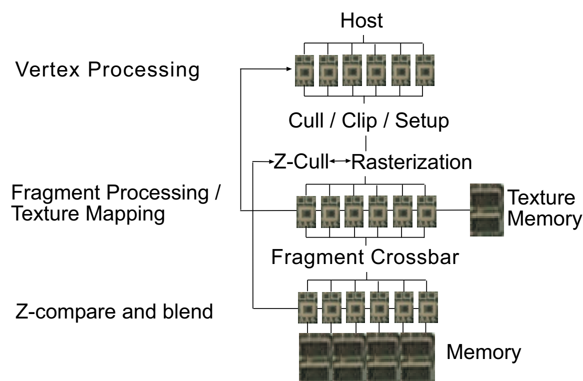

Figure 1: The GeForce 6 series architecture (adapted from [4]).

The input to the graphics pipeline are geometric primitives defined by vertices and its output are the colour values of the primitives' fragments (pixels) which are stored in the video memory of the graphics card. The task of the vertex processing stage is to transform the incoming vertices from object space into screen space and to perform per-vertex calculations such as lighting. The fragment processors compute for every incoming fragment its final colour value using the interpolated vertex attributes from the rasteriser. This pipeline stage has access to global memory in the form of textures which the fragment processors can access in order to calculate fragment colours. In the final stage of the pipeline fragments are blended with the frame buffer or are discarded if they fail optionally applied depth and stencil tests.

2.2 Limitations of GPUs

Programmable GPUs have a higher computational power than CPUs because they are explicitly designed for the simultaneous processing of multiple data-parallel primitives. However, compared to CPUs they offer only a limited instruction set consisting primarily of mathematics operations which are often graphics specific and in general accept as input a limited number of 32-bit floating point 4-vectors. The vertex stage can output a limited number of these floating point vectors, which are interpolated by the rasteriser and passed as input vectors to the fragment stage. Currently the fragment processor can output only 4 floating point 4-vectors, usually representing colours. Each programmable stage has access to global constants and local temporary registers.

Since the write position of a processed fragment is determined in advance by the vertex-parameters and cannot be changed within a fragment program, fragment processors are incapable of performing memory scatter. It is possible to perform memory scatter operations via vertex programs through the recently emerged vertex-texture-fetch capability of current GPUs and the vertex processors' ability to change the target memory address of the coloured fragments. This, however, can lead to memory and rasterisation coherence issues and lower performance [6].

High-level shading languages support traditional C-style explicit flow control constructs. Current GPUs make use of three basic types of data-parallel branching: predication, MIMD (multiple instruction multiple data) branching and SIMD (single instruction multiple data) branching. True MIMD branching is currently only implemented in the vertex processors of the NVIDIA GeForce 6, NV40 Quadro and G70 GPUs. Because of the performance penalties of SIMD and branching by predication, it is important to move the flow-control decisions up to an earlier stage of the graphics

pipeline, where they can be evaluated more efficiently.

3 An Analysis of GPU-Based Volume Segmentation Techniques

The limitations described in subsection 2.2 make it difficult to map volume segmentation algorithms directly to graphics hardware. The instruction count limitation of current vertex or fragment processors can sometimes be bypassed by splitting the computation into several rendering passes and storing intermediate calculation results within textures for subsequent shader executions. Furthermore, hybrid approaches can be used by moving difficult to implement parts of the desired algorithm to the CPU. For example, Lefohn et al. implemented a memory manager for the graphics hardware on the CPU, allowing for the utilisation of sparse data structures for a GPU-based level-set solver [7, 8].

Many successful image segmentation algorithms rely on the dynamic computation of image statistics which are difficult to implement on the GPU. An example is image histograms which require the creation of a memory array containing the number of pixels for every gray-value interval of interest. A simple approach to solve this task is to read each pixel's gray value and increase the corresponding counter in the memory array. This requires the ability to perform data-dependent memory writes, which is a form of memory scatter and, as described previously, is impossible to implement directly in fragment programs. NVIDIA presents a histogram calculation method based on occlusion queries [9].

Another limitation for high-level segmentation algorithms employing additional object information is the available amount of graphics memory. For example, atlas-based image segmentation methods require the construction and access to a database, containing information about object attributes such as shape. As of today, the maximum memory available on consumer PC graphics hardware is 512 MByte, being easily exceeded by the analysis of volumetric and multi-channel imaging datasets.

Algorithms that map well to the parallel graphics hardware architecture have an high arithmetic intensity and are executable in parallel. In 2002 Yang et al. [10] presented a fast threshold based image segmentation method followed by morphological operations to clean and smooth the resulting images. By exploiting the register combiners and blending capabilities of fixed-function consumer graphics hardware, they achieve a speedup by a factor of 5 compared to CPU implementations. Viola et al. [11] adapted the thresholding approach to current graphics hardware and improved it by performing preceding nonlinear image enhancement filters using high level shading languages.

In 2003, Sherbondy et al. [12] presented a nonlinear diffusion-based region growing technique to segment volumetric images. By integrating the filtering and growing process with a direct volume renderer and implementing all three algorithms using vertex and fragment shaders the authors avoid data transfers across the system bus. The resulting speed-up enables the interactive simultaneous segmentation and visualisation of volume data. One common difficulty with edge-based methods is the leak of seeds across weak boundaries which motivates the use of an interactive segmentation process. Observing the growing process and seeing the segmentation results immediately enables the user to select suitable seed points and to adjust the parameters influencing the diffusion process appropriately.

Sherbondy et al. use an ATI Radeon 9800 Pro graphics card and exploit its 3D textures and early z-culling capabilities. They achieve a performance gain of about 10 to 20x over a highly optimised software implementation running on an Intel Pentium 4 2.4GHz based PC. In addition the visual quality of the segmentation results is better than that obtained with threshold-based approaches.

Rumpf and Strzodka [13] introduced a GPU-based solver for a 2D level-set equation with intensity and gradient image-based forces. Lefohn and Whitaker [7] extended this approach to 3D, supporting a more complex user controllable evolution function to reduce the influence of image noise. Subsequently Lefohn et al. presented a sparse 3D level-set solver, which by limiting the computation to only active surface elements provides a speedup of 10-15 times compared to highly optimised CPU solvers [8].

4 Implementation of a hybrid GPU-based Volume Segmentation Technique

We have implemented a hybrid image segmentation technique which uses threshold-based and diffusion-based region growing and utilises sophisticated graphics hardware functionality of consumer level graphics cards. All segmentation algorithms, including image pre-processing as well as morphological segmentation modifiers, have been developed to be executable entirely on the graphics processing unit. The segmentation techniques have been designed and implemented to

be integrated into the object-oriented architecture of the open source visualisation system VTK [14]. Furthermore, a framework application has been developed providing access to volumetric image datasets, an user interface for interactive steering of the segmentation processes, as well as different data visualisations, including volume rendering based on functionalities of the Visualization Toolkit. To provide principle platform independence, OpenGL was chosen as the 3D Graphics API for the implementation of the presented image filtering and segmentation techniques. For the creation of the specific vertex and fragment programs, the high-level OpenGL Shading Language was utilised [15].

4.1 Data Representation and Processing Model

3D image data sets are loaded into the graphics hardware memory as stacks of 2D textures. In order to allow the iterative execution of filter as well as segmentation fragment programs, a second identically structured stack of 2D textures is created. The processing of an image operator is issued by drawing a quadrilateral covering a specific texture slice of the target stack, binding several texture slices from the source stack as input and setting the specific fragment program for execution. Depending on the performed algorithm, one or three textures, i.e. volume slices, will be bound as input for the particular fragment shader. For example, the thresholding operation only requires the data of a single voxel, determines the thresholding result and stores it in the second texture stack at the corresponding voxel location. In contrast, a fragment program performing a region growing step requires voxel information of its spatial neighbourhood and thus needs access to the neighbouring texture slices of the current voxel.

Some operations, like the seeded region growing, require multiple iterations of its computation. Rather than copying the whole frame buffer after each slice has been computed many researchers use pixel buffers which are invisible off-screen rendering targets. The drawback of this pbuffer-technique is, that changing the rendering target usually incorporates an OpenGL context switch, which is an expensive, time consuming function call. Furthermore, this extension is currently platform dependently and only available under the Microsoft Windows operating system.

We exploit the new `EXT_framebuffer_object` OpenGL extension [16] for the realization of the render to texture functionality. It is available in the latest NVIDIA graphics drivers (v77.72) and provides the most efficient way to render to arbitrary textures [17]. The extension allows for the creation of arbitrary frame buffer objects, consisting of a collection of logical buffers, i.e. for colour, depth or stencil values, being usable as texture objects as well as render-buffer objects. Moreover, additional OpenGL rendering contexts are not required and a more efficient management of graphics resources is provided by allowing to share logical buffers between frame-buffer objects.

4.2 Diffusion-based Image Smoothing

In order to improve segmentation results we preprocess image data using a nonlinear edge-preserving image smoothing operator based on an anisotropic diffusion metric proposed by Perona and Malik [18]. The anisotropic diffusion filter reduces noise artifacts while preserving the structure of edges and it hence improves the behaviour of region growing segmentation techniques. The evolution of the image smoothing can be represented by the partial differential equation:

$$\frac{\partial V(t,x,y,z)}{\partial t} = div(g(|\nabla V(t,x,y,z)|)\nabla V(t,x,y,z))$$

where $g(s) = \nu e^{-\frac{s^2}{K^2}}$ and $V(x,y,z)$ is a voxel, t represents an iteration step, and ν and K are user-defined parameters to control the smoothing process.

Following the approach by Sherbondy et al. [12] we discretise the equation using a standard explicit forward Euler approach. Rearranging terms makes it possible to use the dot product for the calculation which is directly supported by the fragment processor hardware and requires only one single clock cycle for its calculation [19].

4.3 Seeded Region Growing

Our segmentation algorithm is based on a seeded region growing approach which allows the iterative application of threshold-based and diffusion-based region growing steps. The seed point selection is a crucial step for the region growing segmentation process. In order to make full use of the parallel nature of modern graphics hardware it is important to select as many seed points as possible. Our manual seed point selection allows the user to interactively select arbitrary regions in the image dataset and thus select multiple voxels as seed points with a user definable seed value. The seed regions can be increased using thresholding and morphological operators.

The threshold-based region growing step is similar to the approach by Viola et al. [11] but we

only consider voxels adjacent to seed voxels for thresholding. The diffusion-based region growing approach follows the method described by Sherbondy et al. [12] and uses a similar equation as described in subsection 4.2. This approach is noticeable slower than the threshold-based region growing method.

In order to combine the advantages of both these methods our approach enables the user to mix threshold-based and diffusion-based region growing steps. The morphological operations erode and dilate can be utilised interactively to provide a user steerable modification of the segmentation result. All methods described in this section have been implemented by fragment shaders and the entire segmentation process is executed on the GPU. More implementation details are described in [19].

5 Application Framework Design

Our application is written in C/C++ and runs on the Microsoft Windows operating systems. It uses the OpenGL graphics library version 2.0 which incorporates the high-level OpenGL Shading Language. Furthermore we use the OpenGL Extension Wrangler Library (GLEW) [20].

We have integrated our GPU-based volume segmentation method into VTK [14] which provides us with interactive user interface tools and advanced visualisation techniques. Figure 2 shows an example of our application at work.

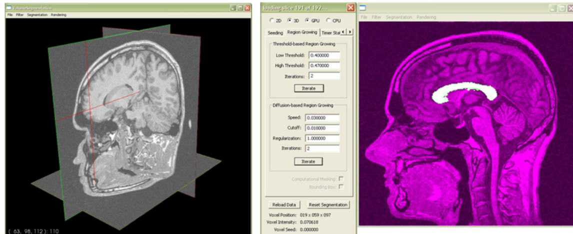

Figure 2: A VTK widget used for seed point selection (left) and the control panel for the interactive segmentation process (right).

6 Results

Our application allows the fast interactive segmentation of large data sets. An example of the results obtained using our interactive hybrid segmentation method is shown in figure 3.

Since our segmentation is performed interactively it is difficult to measure its total performance. Table 1 shows the processing throughput in MVoxels/s for the GPU and CPU implementation of the diffusion-based region growing algorithm:

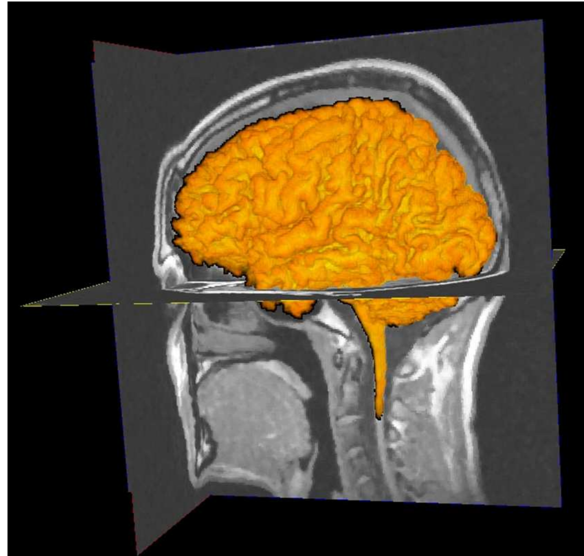

Figure 3: The result of interactively segmenting an MRI data set of the brain.

Size	51×51 $\times38$	85×85 $\times64$	128×128 $\times96$	256×256 $\times192$
GPU	115	150	186	204
CPU	0.66	0.60	0.58	0.54

Table 1: Processing throughput in MVoxels/s for the GPU and CPU implementation of the diffusion-based region growing algorithm.

The measurements indicate the huge speed advantage obtained by using a GPU-based implementation. Similar speed improvements were obtained for anisotropic diffusion filtering, whereas the speed advantage for threshold-based region growing is smaller and varies from a factor of 4 up to 17 for the largest test data set.

Note that the CPU implementation does not exploit additional instruction sets, like MMX, SSE, or SSE2. Therefore, these results are not representative. Nevertheless, the huge performance gain obtained by using a GPU-based implementation can be seen very clearly. The CPU implementation was executed on an 2.4GHz Intel Pentium 4 based PC with 512 MB PC333 DDR RAM. The GPU-implementation was executed on an NVIDIA GeForce 6800 GT graphics card, comprising 256 MB of RAM.

7 Conclusion

We have reviewed and analysed the current state-of-the-art of GPU-based volume segmentation methods and we have explained its fundamental limitations. We have developed an interactive hybrid GPU-based segmentation technique which combines threshold-based and diffusion-based

region growing and we have successfully employed it to segment data sets of up to $256 \times 256 \times 192$ voxels in real time. As with all interactive methods the success of the technique depends a lot on the skills of the user.

Preliminary performance measurements indicate that a GPU-based application is by at least one order of magnitude faster than a comparable CPU-based implementation. In future work we will perform comparisons with optimised CPU implementations and we plan to quantitatively evaluate the quality of our segmentation results.

References

[1] M. Sonka and J. M. Fitzpatrick, *Handbook of Medical Imaging*, vol. 2 - Medical Image Processing and Analysis. SPIE Press, 2000.

[2] J. S. Suri, S. K. Setarehdan, and S. Singh, *Advanced Algorithmic Approaches to Medical Image Segmentation*. Springer Verlag, 2002.

[3] J. K. Udupa and G. T. Herman, eds., *3D Imaging in Medicine*. Boca Raton: CRC Press, 2000.

[4] E. Kilgariff and R. Fernando, *GPU Gems 2*, ch. The GeForce 6 Series GPU Architecture, pp. 471–491. Addison Wesley, 2005.

[5] I. Buck, "GPGPU: General-purpose computation on graphics hardware - GPU computation strategies & tricks." ACM SIGGRAPH Course Notes, 2004.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Proceedings of Eurographics 2005 - State of the Art Reports*, pp. 21–51, Aug. 2005. Dublin, Ireland, August 29 – September 2.

[7] A. E. Lefohn and R. T. Whitaker, "GPU-based, three-dimensional level set solver with curvature flow," technical report uucs-02-017, School of Computing, University of Utah, 2002. URL: `http://graphics.cs.ucdavis.edu/~lefohn/work/rls/gpuLevelSet3D-TR.pdf`.

[8] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "Interactive deformation and visualization of level set surfaces using graphics hardware," in *Proceedings of IEEE Visualization*, pp. 75–82, 2003. 19–24 October 2003.

[9] Nvidia Corporation, "Nvidia software development kit 9.1," 2005. URL: `http://developer.nvidia.com`.

[10] R. Yang and G. Welch, "Fast image segmentation and smoothing using commodity graphics hardware," *Journal of Graphics Tools, special issue on Hardware-Accelerated Rendering Techniques*, vol. 7, no. 4, pp. 91–100, 2002.

[11] I. Viola, A. Kanitsar, and E. Gröller, "Hardware-based nonlinear filtering and segmentation using high-level shading languages," in *Proceedings of IEEE Visualization*, pp. 309–316, 2003. 19–24 October 2003.

[12] A. Sherbondy, M. Houston, and S. Napel, "Fast volume segmentation with simultaneous visualization using programmable graphics hardware," in *Proceedings of IEEE Visualization*, pp. 171–176, 2003. 19–24 October 2003.

[13] M. Rumpf and R. Strzodka, "Level set segmentation in graphics hardware," in *Proceedings of the IEEE International Conference on Image Processing (ICIP 01)*, pp. 1103–1106, 2001.

[14] Kitware, Inc., "VTK Home Page." URL: `http://public.kitware.com/VTK/`.

[15] J. Kessenich, D. Baldwin, and R. Rost, "The OpenGL shading language," 2004. URL: `http://www.opengl.org`.

[16] J. Juliano and J. Sandmel, "EXT_framebuffer_object specification," 2005. URL: `http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt`.

[17] S. Green, "The OpenGL framebuffer object extension," 2005. Game Developers Conference 2005 presentation slides, URL: `http://developer.nvidia.com/object/gdc_2005_presentations.html`.

[18] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[19] S. Schenke, "Analysis and design of GPU-based algorithms for interactive volume segmentation," Diplomarbeit, Friedrich-Schiller-Universität Jena, July 2005. Lehrstuhl für Bildverarbeitung.

[20] M. Ikits and M. Magallon, "The OpenGL Extension Wrangler Library (GLEW)," 2005. URL: `http://glew.sourceforge.net`.