3D Volume Renderization and Connected Component Filtering through Maxtree

André Luís Costa and Wu Shin Ting and Roberto de Alencar Lotufo and Letícia Rittner

Department of Computer Engineering and Industrial Automation, School of Electrical and Computer Engineering, University of Campinas, Brazil
`alcosta@dca.fee.unicamp.br`

December 6, 2013

Abstract

This paper reports the results of a project for 3D MRI data volume visualization using the ray–casting technique implemented in the GPU. The project includes the filtering of connected components organized in a tree structure known as Maxtree. The project was developed over the course IA369E given during the second half of 2013 in the School of Electrical and Computer Engineering, University of Campinas. The produced prototype allow the user to interact with the visualization, to specify custom transfer functions and to filter the data volume. Such functionalities combined give the user the power to enhance and highlight features of interest in the volume being visualized. From learning perspective the results were more than satisfactory, given the knowledge obtained by the student, who is now able to improve and extend the application that can be used in future researches related to the human brain.

1 Introduction

Recent technological advances have made it possible to obtain increasingly large data volumes that may be synthetic from computer simulations, but may also be derived from scans of physical bodies using, for instance, Magnetic Ressonance Imaging (MRI) and Computer Tomography (CT) [1, 2]. Therefore, the visualization of data volumes become one critical task, along with statistical analysis tools, to help scientists understand and gain insight from their data. More over, the interactive visualization is needful to give scientists the ability to map characteristics, select portions, and make the repositioning of the volume, aiming to highlight the desired information.

In this work our goal was to visualize MRI volumes from the human brain with possibility of interaction. Especifically, we employ filters using a maxtree [3] structure to help the selection of desired portions of the volume for visualization. Our application also provide tools for mapping the volume intensity values to custom colors and opacity through transfer functions.

Besides the application goals, and once the project was developed over a course, one goal was also learning. The knowledge of 3D volume renderization techniques and visualization tools is a valuable resource for scientists that works with large data volumes. By having this knowledge the scientist is not only able to explore the data volumes, like using already available visualization tools, but also to incorporate custom functionalities to the application that could be used for specific research purposes.

This paper is organized as follow: Section 2 describes the main tools and methods involved in the project, with some relevant design and implementation details; Section 3 presents the main user interface and relates some usage experiences; in Section 4 a conclusion is presented, along with a perspective for future related works.

2 Tools and methods

The core tools and methods employed in the project are described in this Section. We focus on the techniques chosen and how they interact to give us some desired functionalities like filtering, color mapping and translucency. The whole project was designed to work with OpenGL [4] as the Application Programming Interface (API) to communicate with the Graphics Processing Unit (GPU). Furthermore, we use the OpenGL Shading Language (GLSL) to implement parts of the application that need to run on the GPU.

The application uses the Qt [5] framework to manage the graphical user interface. The main programming language used was Python, with some parts implemented in C/C++.

2.1 Ray–casting

The technique chosen to make the volume visualization, *i. e.*, map a 3D volume to a 2D plane, is known in the literature as *ray–casting* [1, 6]. In this approach the color of every pixel of the 2D image is a composition of the values along a ray that traverse the data volume, perpendicular to the view plane. The ray–casting technique was chosen due to its highly parallelization on the GPU, leaving the CPU available for other operations. Also, it is natural to implement different light interactions with the data [1] once the ray may be considered a ray of light. In our work we implement only the light emission interaction that is simple and sufficient for our purposes.

Its worth to note that there are a variety of techniques to render a 3D volume into a 2D plane. Some of them make an indirect renderization by converting the data content into elements of geometry. These methods were discarded due to the nature of the data we wish to display. Thus, the ray–casting technique was chosen from a set of direct volume renderization methods that include legacy approaches commonly used when the display technology was not yet well developed. One interesting related work published by Westenberg *et al.*[7] also uses the maxtree structure to provide component filtering, however they render the volume using a technique known as *splatting*. The reason we chose ray–casting instead of splatting is that we wanted to solve the renderization problem as much as we could in the GPU. Given that the target data volumes hardly would not fit in the GPU texture memory, a simple ray–casting implementation could do the job. Yet, it still a powerful tool.

The ray–casting implementation is depicted in Fig. 1. As we can see, the main application runs on the CPU and is written mostly using the Python programming language together with a number of libraries that includes Qt and OpenGL. The data volume is loaded from disk with the help of the Nibabel [8] library and preprocessed with Numpy [9] to correct orientation and shape. This process is made only once, after which the resulting data volume is transferred to the GPU texture memory. Then, at each user interaction the rendering variables are updated. That includes the vertices for the volume container and the ray direction in the texture space coordinates relative to the camera view point.

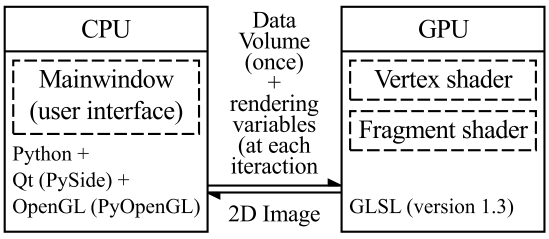

CPU
Mainwindow (user interface)
Python + Qt (PySide) + OpenGL (PyOpenGL)
Data Volume (once) + rendering variables (at each iteraction
2D Image
GPU
Vertex shader
Fragment shader
GLSL (version 1.3)

Figure 1: Diagram showing the main application components and how they interact to implement the ray–casting renderization technique.

The vertex shader is slightly adaptated to transfer to the fragment shader the corresponding texture coordinates at the face point being drawn by each fragment (pixel). The fragment shader is the core implementation of the ray–casting technique. Then, for each fragment the following steps are computed

1. Find the intersections of the ray with the edges of the texture space;
2. From the last intersection to the first, in regards to the camera view point, the volume colors in a set of consecutive equally spaced points along the ray are composed into the final fragment color.

The codes for the vertex and fragment shaders are written in GLSL version 1.3. When the renderization pipeline finishes, it display in the application canvas the resulting 2D image. The transfer function is also applyied in the fragment shader, as described in details in the following Section 2.2.

2.2 Transfer function

The transfer function is the mechanism used in volume rendering to transform raw data values into the optical properties needed to make a picture [1]. Although the transfer function essentially plays the role of a simple color map, it is one of the most important stages of the volume-rendering pipeline as it enable the user to enhance features of interest in the data and can also hide unimportant regions, *i. e.*, it acts like a filter.

Figure 2 show the transfer function editor which was specifically designed for this work, using OpenGL. The editor has two main controlers: in the first the user can edit the color map; and in the second the user can edit the opacity function. Thus, at each interaction on the controlers an unidimensional transfer function with components RGBA is generated with size 256 (once our raw data values are represented by 8 bits). The transfer function is then transferred to the GPU texture memory and the volume renderization is updated.

The renderization update occur in execution time feeding the user who can react and suit the transfer function properly. Editing the opacity component of the transfer function the user can simulate a thresholding of the volume leading to the renderization of an isosurface, and can also make the volume translucent.

2.3 Component filtering

Component filters works not only with based on the raw values of each of the data volume elements, but also based on how they are connected. One efficient way to implement component filters is to work with a tree structure

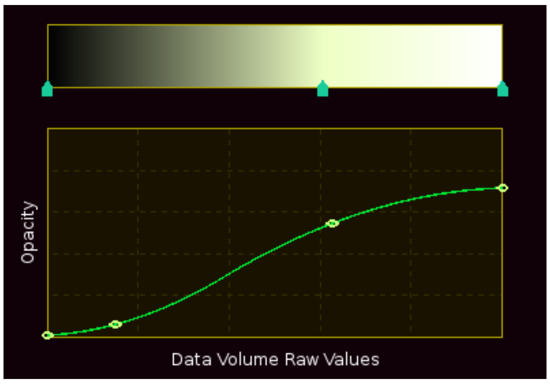
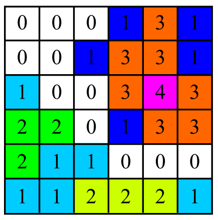
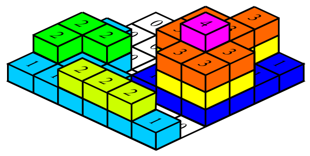


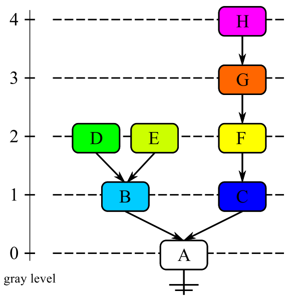
Figure 2: Transfer function editor that enable the user to specify a completely custom color map.



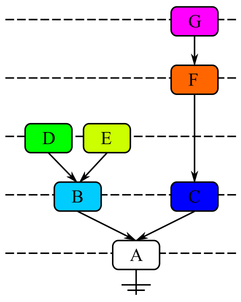
(a) f



(b) f in 3D



(c) Component Tree



(d) Maxtree

Figure 3: Component tree and maxtree representations from a gray level 2D image f.

known as Component Tree (CT), which is an hierarchy of connected components at distinct thresholds of an image [10]. Let f be a gray scale digital image. There exists one possible threshold

$$f_k = \begin{cases} 1, & \text{if } f(x) \geq k \\ 0, & \text{otherwise} \end{cases}, \tag{1}$$

for each gray level $k \in \{f(x)\}$, where $x \in f$. We call f_k a section k of f. There is a relation of inclusion between connected components at distinct image sections. The component tree is built based on this observation. Figure 3 show a gray level 2D image f (Fig. 3(a)) and its 3D representation (Fig. 3(b)) if we take the gray level as the pixel altitude. It is observable that each level k has a number of connected components, also called level components, that are contained on the level components from levels $i < k$. Thus, we can build the component tree ilustrated in Fig. 3(c).

Maxtree is also a tree structure of level components like the CT. However, it has a simpler representation and less nodes. Note that the level component F from the component tree represented in Fig. 3(c) is not present in the maxtree of the same image f, that is being represented in Fig. 3(d). Although a maxtree construction is in general easier and faster to build than a CT, it requires some atention in the implementation of certain component filters.

In our project we use a maxtree representation of the data volume. The implementation was made in the programming languages C/C++ (tree assembling) and Python (representation conversion and filtering by volume), and was inspired on the algorithm proposed by Salembier [11], which can build the maxtree in linear time.

The implemented filter disable the maxtree nodes according with the volume measures for its associated level components. The maxtree nodes whose level components has volume within a range between a minimum and a maximum values specified by the user are allowed to be rendered. After each application of the component filtering the data volume is reconstructed and transferred to the GPU texture memory, updating the visualization.

3 Visualizing

In this Section we present the application user interface and report some usage experiences. Figure 4 show the user interface and some MRI data volume renderizations produced by the developed application. Notice that there is a custom color map and opacity function applyied in the renderization product. One observable effect is translucency.

The application was designed to run in the current personal computers. Therefore, when the user is interacting with the visualization the number of samples from a ray is reduced. When the interaction ends, the normal rendering pipeline is restored. Our test system was a laptop with a processor multicore Intel Core i3-2330M CPU @ 2.20GHz $\times$ 4 cores, a embedded GPU Intel Sandybridge Mobile x86/MMX/SSE2 and 3 gigabytes of working memory. The Operating System was the Ubuntu 13.04 32 bits. A quite humble machine for graphical purposes, but still was able to run the application smoothly with good response time for the user to feel the interactivity with the camera and the transfer function operations.

The component filtering by volume using the maxtree took about 2 seconds to rebuild the data volume and transfer it to the GPU. Given the application nature and the frequency by which a component filtering is required, this time has virtually no impact in the user interactivity. Although the maxtree building took about 30 seconds, this process is made only once when the data volume is loaded from disk. These performance information were obtained by measuring the system time took for each operation for a test data volume with dimensions $120 \times 224 \times 224$, *i. e.*, about 6 million voxels, that led to a maxtree with about 170,000 nodes.

4 Conclusion

The application developed in the course allow the user to rotate the volume, specify a custom transfer function and filter components by volume. These controls give the user a wide range of possibilities and can be extended in future versions. Although the application is still a prototype with lack of many functionalities, we consider it a successful result for the course, specially in regards to the knowledge and experience obtained.

The prototype application can be used for further studies and can evolve to a fully functional application that could be used by medical professionals and scientists to analyse MRI and CT data volumes. We do expect to improve the application ability to enhance and highlight different portions of brain data volumes by implementing other component filters through the maxtree, and also by using automatic and interactive segmentation tools.

References

[1] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf, *Real-Time Volume Graphics*. A K Peters, 2006.

[2] M. Friendly, *Handbook of Computational Statistics: Data Visualization*. Springer–Verlag, 2007, vol. III, ch. A Brief History of Data Visualization, pp. 1–34.

[3] E. Carlinet and T. Géraud, *Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2013, ch. A comparison of many maxtree computation algorithms, pp. 73–85.

[4] Official Site, "Opengl," last access in dec/2013. [Online]. Available: http://www.opengl.org/about/

[5] ——, "Qt gui framework," last access dec/2013. [Online]. Available: http://qt-project.org/

[6] T. Akenine–Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd ed. A K Peters, 2008.

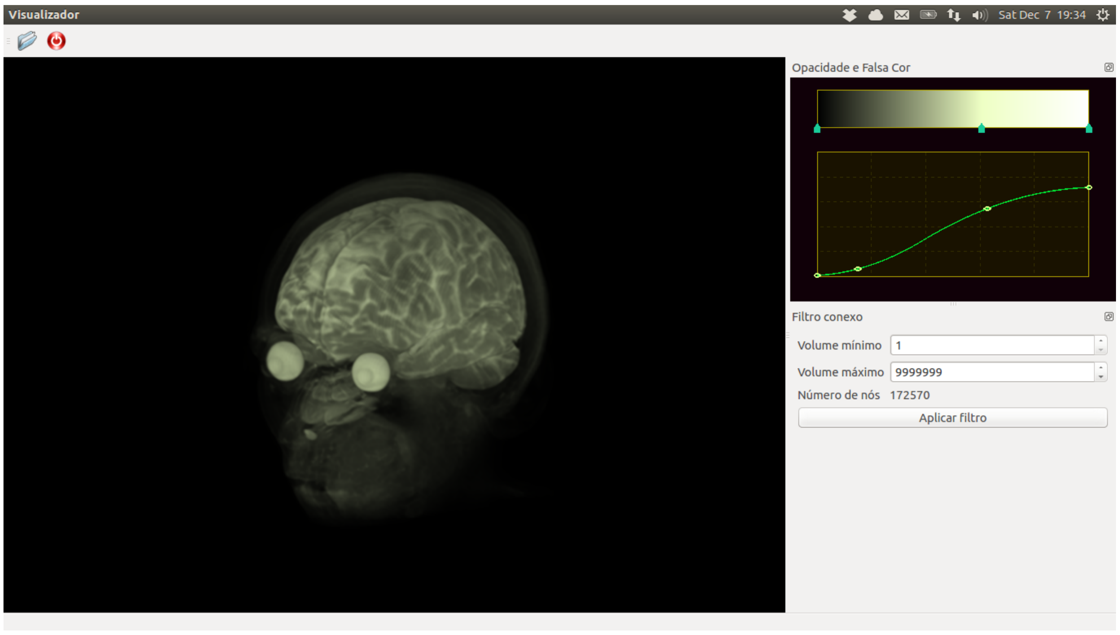
[7] M. A. Westenberg, J. B. T. M. Roerdink, and M. H. F. Wilkinson, "Volumetric attribute filtering and interactive visualization using the max-tree representation," *IEEE Transactions on Image Processing*, vol. 16, no. 12, pp. 2943–2952, 2007.

[8] Official Site, "Nibabel library," last access in dec/2013. [Online]. Available: http://nipy.org/nibabel/

[9] ——, "Numpy: matricial operations in python," last access in dec/2013. [Online]. Available: http://www.numpy.org/

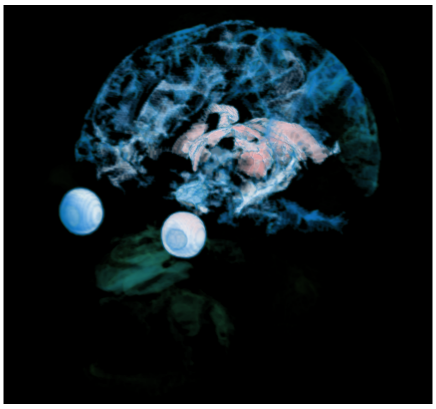
[10] V. Mosorov and T. M. Kowalski, "The development of component tree structure for grayscale image segmentation," in *Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2002. Proceedings of the International Conference*, 2002, pp. 252–253.

[11] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, 1998.

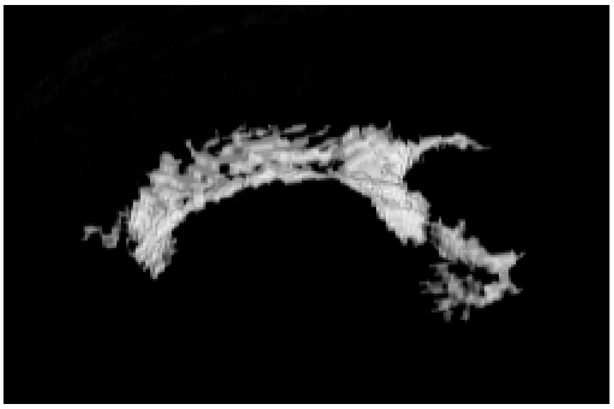


Volume mínimo 1
Volume máximo 9999999
Número de nós 172570

(a) Main user interface



(b) Component filtering with custom transfer function



(c) Internal brain structure being highlighted by component filtering

Figure 4: User interface and renderizations using component filtering and custom transfer functions.