An evaluation of the Volume on Surface (VoS) approach

Rafael Silva Santos, Danilo Medeiros Eler, Rogério Eduardo Garcia, Ronaldo Celso Messias Correia
UNESP - Univ Estadual Paulista, Faculdade de Ciências e Tecnologia
Presidente Prudente - SP, Brazil
rafael.silva.sts@gmail.com, {daniloeler, rogerio, ronaldo}@fct.unesp.br

***Abstract*—Hybrid volume rendering algorithms combine techniques of different categories of volume visualization. Volume on Surface (VoS) is a hybrid volume rendering technique that maps volume information to isosurfaces, intending to accelerate the volume rendering process. In this paper, we introduce an improved version of the VoS technique. Furthermore, we conduct an evaluation of CUDA-based versions of both the original and the improved approach, comparing them with the conventional Ray Casting algorithm. In some results, our technique is more than eight times faster than Ray Casting and thirty times than original version. In addition to the main discussion, we investigate how a surface simplification influences the performance and the quality of the images rendered by our approach.**

***Keywords*-Volume on Surface (VoS); hybrid volume rendering; volume rendering**

I. Introduction

Volume visualization consists of a set of computer graphics techniques that allows exploring the interior of volumes [1]. These techniques provide working tools for different areas of science, such as Chemistry, Geology and Medicine [2].

A great challenge in volume visualization is the high computational cost of the volume rendering process [3]. Since the last two decades, new hardware technologies are increasingly used to cope with such issue [4]. However, even before the hardware advances, researchers have looked for alternatives to reduce the execution time in volume rendering. A set of these alternatives is the Hybrid Volume Rendering (HVR) techniques [5]. An HVR technique combines two or more approaches of volume visualization, in order to achieve a performance increase [5, 6] or to group any advantages of the approaches [3].

An example of HVR technique is the **Volume on Surface** (VoS) approach [3]. VoS combines Ray Casting with an isosurface, affording the visualization of volume details through this structure. The technique aims to accelerate the rendering process.

In this paper, we evaluate the VoS approach. First of all, we present an improved version of the VoS technique. Compared to the original technique, our approach has the following advantages: higher quality images, less memory usage and a performance increase when the transfer functions are adjusted. In this document, we use the term VoS^M to refer to the original technique and the term VoS^{M*} to our version. Moreover, whenever we mention the term VoS, we are referring to both techniques.

For the evaluation, we implement CUDA versions of both VoS approaches and compare these techniques with a CUDA-based Ray Casting. Our comparison focuses on the following aspects: quality of the rendered images and computational cost (execution time and memory usage). In some results, the VoS^{M*} spends 81.61% less runtime than Ray Casting. Apart from the comparison, we investigate how a surface simplification influences the performance and the quality of the images generated by the VoS^{M*} approach. We use a filter available in the Kitware ParaView software to simplify the isosurfaces that are used to perform our experiment.

Besides this introduction, the reminder of this paper is organized into other five sections. Related works are presented in Section II. We present the VoS^M approach and introduce our improvement in Section III. Our strategy of implementation is discussed in Section IV. The results are reported in Section V. Finally, Section VI concludes this paper.

II. Related works

A. Ray Casting

Ray Casting is probably the most popular and adapted Direct Volume Rendering (DVR) algorithm in the literature [1, 7]. Briefly, the algorithm consists of a rays shooting from the image plane toward the volume. Initially, it is defined an image plane between the observer and the volume. Next, one ray is shot from each pixel of the image plane. The algorithm accumulates contributions of color and opacity of each volume element (cell or voxel) along a ray path. When a ray targets the end of the volume, or the accumulated opacity reaches a maximum value, the path is interrupted. At last, the color accumulated in the ray is assigned to the source pixel.

Levoy [8] proposed a typical model that describes the color and opacity accumulations along a ray path. This model is presented below:

$$\begin{aligned} \alpha_{out} &= \alpha_{in} + \alpha_s(1-\alpha_{in}) \\ C_{out} &= C_{in} + C_s\alpha_s(1-\alpha_{in}) \end{aligned}, \tag{1}$$

where, C_{in} is the accumulated color and α_{in}, accumulated opacity. C_s and α_s are results of the transfer functions applied to the sample. Finally, C_{out} is the accumulated color and α_{out}, the accumulated opacity after the accumulation.

Over the years, different approaches of the Ray Casting algorithm have been proposed. Weinlich et al. [9] compared a CUDA-based Ray Casting with the OpenGL implementation

of this algorithm. Maršálek et al. [10] developed a CUDA-based approach that uses interpolation and texture memory to improve the performance of the algorithm. Zhao et al. [11] proposed another approach that also uses interpolation and texture memory on CUDA. However, this approach use Sobel filter to calculate gradients and provides a better quality and performance. Later, Bethel and Howison [12] introduced a technique that focuses on the memory access scheme for to improve the performance of the parallel Ray Casting on CPU and GPU.

B. Hybrid volume rendering

The term "hybrid volume rendering" (HVR) describes a volume visualization strategy that integrates different approaches of rendering, aiming to group advantages of each approach. Typically, an HVR technique combines two or more techniques of Direct Volume Rendering (DVR) and Surface-Fitting (SF) [3, 5].

Levoy [13] proposed one of the first HVR techniques. The approach combines Ray Casting and SF, providing simultaneous display of both volume and surface data. Tost et al. [14] introduced another approach with the same goal, but that uses the Splatting algorithm. Later, two techniques [15, 16], quite similar to VoS, were proposed. The techniques integrate a texture mapping and isosurfaces to afford the visualization of volume details.

Marroquim et al. [17] introduced the Projected Tetrahedra with a partial pre-integration (PTINT) algorithm. PTINT mixes DVR with SF and is divided into four steps. In the first step, PTINT projects the tetrahedrons in fragment shaders. Next, a volume rendering process is associated with each primitive, i.e., a vertex of the tetrahedron surface (composed of tetrahedrons). In the third step, a final color is estimated by a partial pre-integration. The last step renders the final images and ends the algorithm. Later, Maximo et al. [18] introduced an adaptation of the PTINT, which uses rasterization to allow that the entire algorithm is performed on GPU. Therefore, the adapted version [18] decrease the execution time.

Most recently, Liang et al. [6] proposed an HVR GPU-based Ray Casting framework that combines DVR with spherical texture maps. This framework focuses on an acceleration of the volume rendering process with large atmospheric datasets. The approach limits the data, using only a part of the original dataset. Moreover, this portion of data is even interpolated to accelerate the rendering process.

III. Volume on Surface

A. VoS^M

VoS^M (Volume on Surface) [3] is an HVR technique, which allows the visualization of volume details through an isosurface extracted from this volume. In high-level, the approach can be described as a combination of the Ray Casting algorithm with a surface rendering process. So it is important to note that VoS^M is not a DVR technique, but an approximation that aims to accelerate the visualization process.

In the VoS^M algorithm, the volume data is arranged in a regular grid. The isosurface is a polygon mesh. A pipeline of the VoS^M is shown in Fig. 1. The algorithm is divided into four steps: cones construction, pre-visualization, vertex color assignment and lighting & projection.

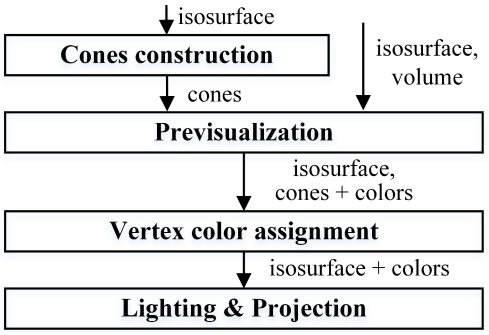


Fig. 1: VoS^M pipeline.

In the first step, the algorithm constructs a set of cones, with different opening angles, on each vertex of the isosurface. Each cone has a set of rays that are equally spaced. Fig. 2 shows an example of virtual cones attached to a vertex. The normal vector (e.g., see the blue arrow in Fig. 2) of each vertex is the central axis of its cones. After the **cones construction** step, there are n rays associated with each vertex, where $n =$ cones per vertex $\times$ rays per cone.

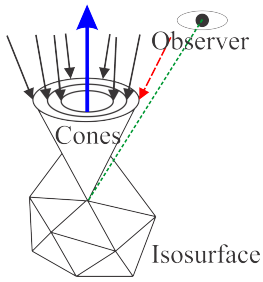


Fig. 2: Example of cones structure in the VoS^M algorithm. Three virtual cones attached to an isosurface vertex (Adapted from [3]).

The **previsualization step** comprehends a volume rendering process. This step is similar to the Ray Casting algorithm. The rays are shot from each vertex toward the volume. Color and opacity contributions of each reached voxel are accumulated along a ray path, as defined in Equation 1. At the end of the process, a final color is assigned to the ray. Thus, there are n colors associated with a vertex.

The last two steps constitute a surface rendering process. In the **vertex color assignment** step, a single ray is chosen to assign a color to a vertex that is visible from the observer's position. The algorithm selects the ray that is the closest to the observation line (e.g., see the dashed green line in Fig. 2). A dot product computation may solve the problem of finding the closest ray. Initially, it is considered $R = \{r_1, ..., r_n\}$ as a set of unit vectors that represents the rays associated with a vertex

and l as the unit vector that represents the observation line. The selected ray is one that the dot product, $r_i \cdot l$, produces the largest positive value [3]. At last, any traditional rendering algorithms may perform the **lighting** & **projection** step.

The VoS^M technique assumes that a user may interact in two different ways during the visualization. If the observer's position is modified, it is necessary to perform again just the surface rendering process (vertex color assignment and lighting & projection). On the other hand, if the transfer function is adjusted, the previsualization and surface rendering process must be performed again.

B. VoS^{M*}

VoS^{M*} is an improvement of the VoS^M technique. Our approach has the following goals: to improve the quality of the rendered images, to decrease the memory usage and to reduce the execution time when a transfer function is adjusted.

The VoS^{M*} algorithm does not use cones to guide the rays shooting. A single ray is shot from the observer's position toward the vertex. The direction of this ray coincides with the observation line, as shown in Fig. 3.

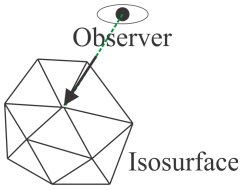


Fig. 3: Example of a ray shot from an isosurface vertex in the VoS^{M*} algorithm (Adapted from [3]).

Fig. 4 shows the VoS^{M*} pipeline. The volume and isosurface have the same structures as those used by the VoS^M. Nonetheless, the VoS^{M*} algorithm comprises just two steps. In the **previsualization** step, a single ray is shot from each vertex. At the end of this step, the color accumulated in the ray is assigned to the source vertex. Similarly to VoS^M, any traditional **lighting** & **projection** algorithms may render the final image. Regardless of the user interaction, it is always necessary to perform the two steps again.

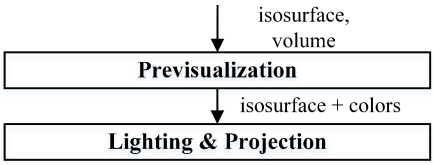


Fig. 4: VoS^{M*} pipeline.

IV. Implementation

It is not difficult to note that the VoS techniques, as the Ray Casting [8], are intrinsically parallel. Thus, we use C/C++, CUDA and OpenGL to parallelize these algorithms.

Before introducing the implementation of techniques, we describe the data structures. The **volume** is a regular grid of intensity values that is implemented as an array of bytes. The **isosurface** is a triangle mesh. Four arrays constitute the isosurface structure. A floating-point array stores the vertices, and another array stores the normal vectors of such vertices. Both vertices and their normal vectors are 3D coordinates. A third array stores indexes to positions of the array of vertices. Each three consecutive positions in this array represent a triangle. At last, a byte array stores an RGB color for each vertex. In the **cones** structure, a ray consists of a 3D coordinate and an RGB color. Thereby, the cones structure is composed by two arrays: a floating-point array for 3D coordinates, and byte array for colors (RGB color).

A. VoS techniques

The **VoS**^{M} algorithm is organized into four steps. A first CUDA kernel implements the cones construction step. This kernel has as input parameters: the surface, the number of rays per cone and the opening angles. In this kernel, each thread build a virtual ray, still a color assigned. A second CUDA kernel performs the previsualization step. Each thread shot one ray from the vertex toward the volume. After the end of a ray path, the accumulated color is assigned to the ray. Besides the volume and the cones, the kernel also has the transfer functions as input parameters. The last CUDA kernel implements the vertex color assignment. Each thread finds the nearest ray to the observation line and thereby, the color that must be assigned to each vertex. The lighting & projection step is implemented in OpenGL. The algoritms of this library are also implemented on GPU. Therefore, both CUDA and OpenGL must access the same data. To afford this, we use CUDA-OpenGL interoperability. Thus, the surface structure is implemented using Vertex Buffer Object (VBO) [19].

In the implementation, we consider two situations of user interaction, as described in Section III. In VoS^M, if the transfer functions are adjusted, the first and the second kernels must be performed again. On the other hand, if the user modifies the position of the observer, only the third kernel is performed. Still regarding the user interaction, a GUI, on the host (CPU), handles the transfer function adjustments and migrates this data to the device (GPU). The transfer functions are implemented in Look-Up-Tables (LUTs).

A single CUDA kernel implements the parallel part of the **VoS**^{M*} algorithm. This kernel performs the previsualization step and has the following input parameters: surface, volume, transfer functions and observer's position. In contrast to the VoS^M algorithm, a same thread performs the ray shot and assigns the final color to the isosurface vertex. The lighting & projection step is implemented in the same way that in the VoS^M. The implementation of the GUI and the transfer functions are also the same. In the VoS^{M*} approach, the kernel must always be performed for any user interaction,.

B. Ray Casting

We implement the conventional back-to-front Ray Casting in a single CUDA kernel. In such kernel, each thread shoots one ray from each image plane pixel toward the volume. The composition model is established in Equation 1.

Similarly to the VoS techniques, the transfer functions are implemented as LUTs. Beyond this, the Ray Casting implementation also takes advantage of the CUDA-OpenGL interoperability. The image plane (matrix of pixels) is stored in a Pixel Buffer Object (PBO) that is shared between CUDA and OpenGL.

V. Results

A. Experimental Design

In the tests, we use four volume datasets. The human head and chest datasets are available at the University of Erlangen-Nuremberg Datasets Repository[1]. The stented abdominal aorta is provide by the VolVis Repository[2]. Finally the foot is available with the VolView software. The specification of the volume datasets is presented in Table I.

TABLE I: Datasets specification.

Datasets	Dimensions
Chest	$384 \times 384 \times 240$
Foot	$102 \times 247 \times 200$
Head	$128 \times 256 \times 256$
Stented abdominal aorta	$512 \times 512 \times 174$

Table II presents the specification of the isosurface extracted from each volume dataset.

TABLE II: Isosurface meshes specification.

Source datasets	Vertices	Polygons
Chest	732158	1452730
Foot	120720	238816
Head	274896	544624
Stented abdominal aorta	1331004	2641370

We run all tests on a computer with the following configuration: Intel Core 2 Quad Q6700 CPU, 4 GB RAM, Windows 8.1 64 bits, CUDA 7.5 SDK and NVIDIA GeForce GT 740 GPU with 1 GB RAM. Moreover, the observer's position and any scene parameters are the same for the each dataset.

The isosurfaces are extracted by Kitware VolView 2.0 software and converted into a VTK file (vtkPolyData file format) by Kitware ParaView 5.0 software. Unless otherwise specified, the display window for all tests has the size of 800×600 pixels.

In the VoS^M technique, all tests are performed with the following settings: four rays per cones and three cones per vertex. Besides the following opening angles of the cones: $20°$, $45°$ and $70°$.

B. Image quality comparison

Fig. 5 shows images of a human chest, foot and head. From this figure, we can observe that the quality of the images produced by VoS^M are not satisfactory. In these images, there is a great loss of volume details. On the other hand, the images rendered by VoS^{M*}, despite a little loss of detail, are similar to those rendered by the Ray Casting algorithm. However, it can be observed some distortions of the volume structures in our technique as well in VoS^M. These distortions are results of the visualization trough the isosurface, since the observer's position is the same for each test. We emphasize that the VoS approaches are not DVR techniques, but approximations to them.

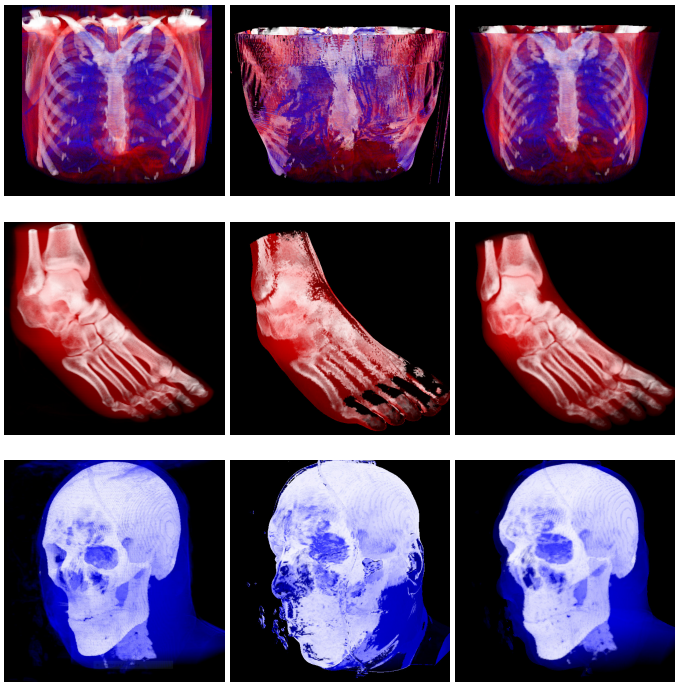


Fig. 5: Images rendered by Ray Casting (first column), VoS^M (second column) and VoS^{M*} (third column). From top to bottom: datasets of a human chest, foot and head.

In the tests presented here, we do not change the setting of rays and cones by the VoS^M technique. Nevertheless, it is important to mention that an increase in the number of rays per vertex can improve the quality of the images rendered by this technique [3]. In a hypothetical best case, at least one ray at each vertex coincides with the observation line. In this case, the images rendered by VoS^M would be like than those generated by our technique.

C. Memory usage comparison

Table III shows the estimated memory usage of each technique. We estimate the memory usage regarding structures described in Section IV. The storage of additional structures, such isosurfaces, becomes the memory usage in the VoS techniques higher than in Ray Casting. Moreover, VoS^M presents a higher memory usage than VoS^{M*}, since this technique also stores a structure of cones. The memory usage of VoS^M is, on average, 534% higher than in Ray Casting. In turn, VoS^{M*} consumes 176.23% more memory than reported by Ray Casting.

[1]University of Erlangen-Nuremberg Datasets Repository. Available at: http://www9.informatik.uni-erlangen.de/External/vollib/. Accessed: July 10 2015.

[2]VolVis Repository, University of Tübingen. Available at: http://volvis.org/. Accessed: July 10 2015.

TABLE III: Estimated memory usage in MB.

Datasets	VoS^M				VoS^{M*}			Ray Casting
	Volume	Isosurface	Cones	Total	Volume	Isosurface	Total	Volume
Chest	33.8	35.5	126.9	196.2	33.8	35.5	69.3	33.8
Foot	4.8	5.8	20.9	31.5	4.8	5.8	10.6	4.8
Head	8	13.3	47.7	69	8	13.3	21.3	8
Stented abdominal aorta	43.5	64.5	230.7	338.7	43.5	64.5	108	43.5

From the Tables I and II, it is easy to realize that the memory usage is proportional to the volume dimension and the size of the isosurface mesh. In VoS^M, the number of rays and cones also increases this usage.

D. Performance comparison

We establish two situations of user interaction to evaluate the execution time of the techniques. In the first situation, we rotate the position of the observer around the y-axis, simulating a modification of the observer's position. In the second situation the transfer functions are adjusted during this rotation.

Fig. 6 shows the performance results for the first situation. Both VoS techniques have better performance than Ray Casting for all datasets. Among the two techniques, VoS^M is the fastest. On average, VoS^M spends 93.26% less runtime than Ray Casting. In turn, VoS^M spends 81.61% less runtime than Ray Casting. On regards the frame rate (FPS), VoS^M is, on average, 1452.08% higher than Ray Casting, VoS^{M*} is 725.27% higher.

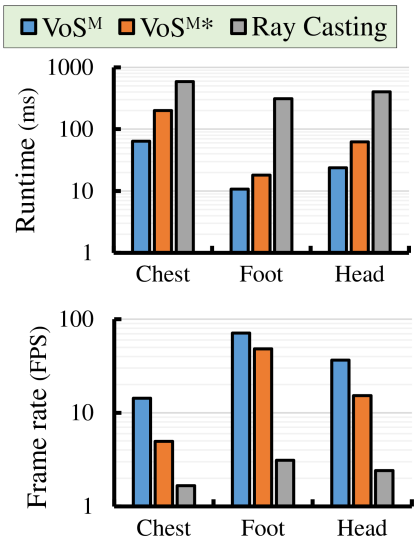


Fig. 6: Runtime and frame rate (FPS), both in $\log_{10}$ scale, of the Ray Casting, VoS^M and VoS^{M*} during a rotation of the observer position around the y-axis.

We highlight that the results presented in Fig. 6 correspond to the best case of VoS^M technique. Whereas just the vertex color assignment and lighting & projection are performed, which explain the results obtained.

The instructions performed on the kernels of both Ray Casting and VoS^{M*} are very similar. However, there is a discrepancy in the performance of these approaches. We tend to think that this variation is related to the number of rays shot. This number corresponds to the size of the display window in the Ray Casting ore the number of visible vertices in the VoS^{M*}. To better investigate this discrepancy, we conducted a further comparison among the techniques, varying the size of the display window.

Fig. 7 reports the runtime and frame rate of the techniques when the transfer functions are adjusted. Ray Casting and our technique have the same runtimes and frame rates than the presented in Fig. 6. In contrast, VoS^M has a great performance decrease. This situation represents the worst case of the technique. In this case the steps of previsualization, vertex color assignment and lighting & projections must be performed. On average, VoS^M spends 585.44% more time than Ray Casting and the its frame rate (FPS) is 71.02% lower.

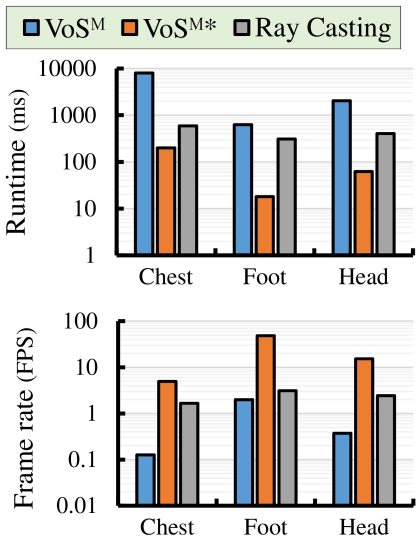


Fig. 7: Runtime and frame rate, both in $\log_{10}$ scale, of the Ray Casting, VoS^M and VoS^{M*} during an adjustment of the transfer functions and a rotation of the observer position around the y-axis.

E. Discussion and an additional comparison among VoS^{M*} and Ray Casting

Based on results so far, we can assert that is not feasible to use the VoS^M technique. Although VoS^M to be faster when just the position of the observer is modified, the technique

becomes much slow if the transfer functions are adjusted. Moreover, the quality of the images rendered by VoS^{M} is not satisfactory and the memory usage is large. Regarding the VoS techniques, we point out that the performance and the quality of images are not dependent on the window resolution.

On the other hand, VoS^{M*} showed promise. The technique proved that the isosurface extracted from a volume can be used as a resource to accelerate the rendering process. Although in comparison to the Ray Casting, the VoS^{M*} has a larger memory usage and renders images with a little loss of detail. We have already mentioned that it is a good guess to assume that the number of rays is responsible for the performance difference among Ray Casting and VoS^{M*}. Thereat, it is also important to think that Ray Casting can be the fastest in any particular case. To bring out this discussion, we perform an additional test with a larger volume dataset, the human stented abdominal aorta. In this test, we also resize the display window of the Ray Casting to 600×300 to reduce the number of rays shot and thus to increase the Ray Casting performance.

Fig. 8 shows the runtime and frame rate of the VoS^{M*} and Ray Casting with display windows resized to 800×600 (RC-1) and 400×300 (RC-2) pixels. In this test, RC-2 has the best performance. RC-2 spends, on average, 55.36% less runtime than VoS^{M*} and 87.3% than RC-1. Regarding the frame rate, the FPS obtained by RC-2 is, on average, 117.38% higher than VoS^{M*} and 640.59% than RC-1.

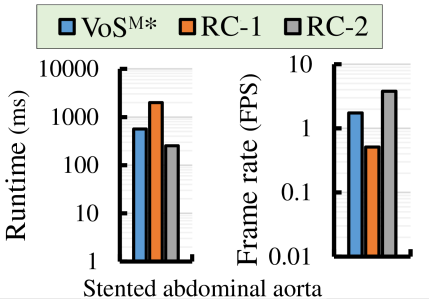


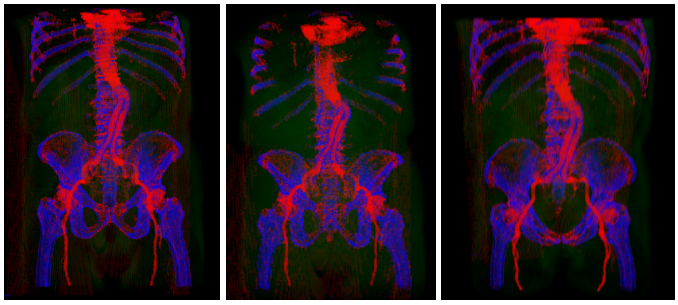
Fig. 8: Runtime and frame rate, both in $\log_{10}$ scale, of the VoS^{M*}, Ray Casting with display window size of 800×600 (RC-1) and 400×300 pixels (RC-2) during a rotation of the observer position around the y-axis.

The images generated by VoS^{M*} and Ray Casting is shown in Fig. 9. Despite the performance gain, we can observe that the quality of the image rendered by RC-2 is lower those generated by VoS^{M*} and RC-1.

We will not show any results, but it is impractical to use VoS^{M} technique to render the dataset of the human stented abdominal aorta. If there is an adjustment of transfer functions, we estimate that the approach would spend more than 10 minutes during each user interaction.

F. Simplification of surfaces

In the last tests, we investigated how a surface simplification influences the performance of the VoS^{M*} technique. The surfaces are simplified using the decimate filter of the Kitware



(a) Ray Casting with 800×600 pixels (b) Ray Casting with 600×300 pixels (c) VoS^{M*}

Fig. 9: Images of a human stented abdominal aorta rendered by VoS^{M*} and Ray Casting with the following display window sizes: 800×600 and 600×300.

Paraview Software with the following targets of reduction: 25%, 50%, 75% and 90%.

The decimation filter can produce holes in the surfaces[3]. So, to cope with this issue, the software provides a "preserve topology" option. This setting does not guarantee that the preset value of reduction is achieved, but decreases the possibility of generation of holes in the surface.

The memory usage for the decimated isosurfaces is presented in Table IV. On average, a decimation of 25% reduces the memory usage by 25%; the decimation of 50%, by 49.9%; the decimation of 75%, by 74.9% and the decimation of 90%, by 79.6%.

TABLE IV: Estimated memory usage (MB) of the decimated isosurfaces in the VoS^{M*} technique.

Datasets	**Decimation factor**				
	0%	25%	50%	75%	90%
Chest	35.5	26.6	17.8	8.9	6.1
Foot	5.8	4.4	2.9	1.5	1.3
Head	13.3	10	6.7	3.3	3.2
Stented abdominal aorta	64.5	48.4	32.3	16.1	11.7

Fig. 10 reports the runtime and frame rate of the VoS^{M*} for the isosurfaces without any reduction and with the following decimation factors: 25%, 50%, 75% and 90%. From this figure, it is possible to note that the decimation of isosurface provides a performance increase of the VoS^{M*}. On average, a decimation of 25% afford a reduction of 22.24% in the runtime; a decimation of 50%, 45.77%; a decimation of 75%, 71.80% and a decimation of 90%, 76.72%. Regards the frame rate, a decimation of 25% produces an increase of 26.67%; a decimation of 50%, 77.40%; a decimation of 75%, 220.1% and a decimation of 90%, 297.35%.

In spite of a performance increase and less memory usage, the simplification of a surface can decrease the quality of

[3]Information obtained from the documentation of the Paraview 5.0 software. Available at: http://www.paraview.org/ParaView/Doc/Nightly/www/py-doc/paraview.simple.Decimate.html. Acessed: 10 March 2016.

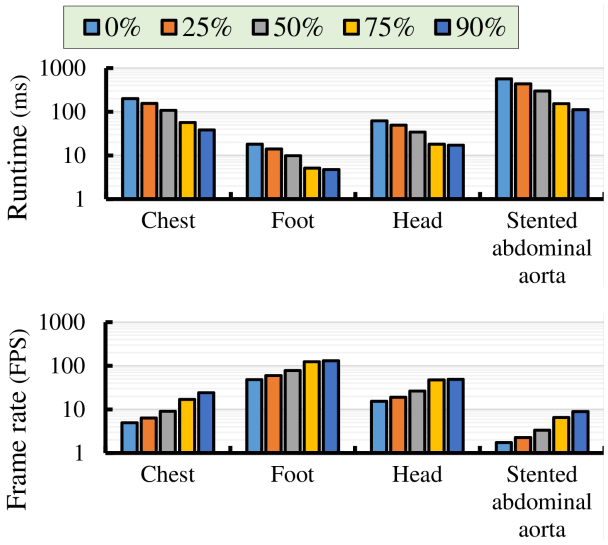


Fig. 10: Runtime and frame rate, both in $\log_{10}$ scale, of the VoS^{M*} for the isosurfaces without any reduction (0%) and with the following decimation factors: 25%, 50%, 75% and 90%.

the images rendered by VoS^{M*}. Fig. 11 shows the images generated with decimated surfaces. The images generated from an isosurface with the reduction factor of 25% are comparable to the images obtained from the isosurfaces without any simplification. However, the images rendered from the isosurfaces with reductions of 75% and 90% have a much lower quality. Although, we can confirm that the surface simplification is a promissory resource to increase the performance of the VoS^{M*}.

VI. Conclusion

In this paper, we presented an improved version of the VoS technique [3]. Moreover, we evaluate the behavior of both the original version of VoS and our approach in a CUDA implementation. From the results, we can assert that is not feasible to adopt the VoS^{M} technique. On the other hand, our approach shows promise. Besides all, we can also affirm that the performance of VoS^{M*} can be increased by a surface simplification. Although, we emphasize that the VoS^{M*} is not a substitute for Direct Volume Rendering techniques. Our technique is an alternative to situations in which a performance increase is preferable over the generation of high-quality images. For example, when VoS^{M*} can be adopted for rendering previews during a user interaction.

Different reasons justify the choice of the Ray Casting technique as a comparison parameter. First, the Ray Casting algorithm is relatively simple to implement. Second, the images rendered by such technique have a high-quality. Finally, the Ray Casting is the volume rendering algorithm most used and discussed in the literature. However, the use of other hybrid volume rendering techniques as comparison parameters is a limitation of our research and a potential future work.

Acknowledgment

This work was partially supported by FAPESP (State of São Paulo Research Foundation) grants (2015/00622-7).

References

[1] T. T. Elvins, "A survey of algorithms for volume visualization," *SIGGRAPH Comput. Graph.*, vol. 26, no. 3, pp. 194–201, Aug. 1992. [Online]. Available: http://doi.acm.org/10.1145/142413.142427

[2] A. E. Kaufman, "Volume visualization: Principles and advances," 2003.

[3] D. M. Eler, P. S. H. Cateriano, L. G. Nonato, M. C. F. Oliveira, and H. Levkowitz, "Empowering iso-surfaces with volume data," *Proceedings of GRAPP International Conference on Computer Graphics Theory and Applications*, 2006.

[4] J. Beyer, M. Hadwiger, and H. Pfister, "A survey of gpu-based large-scale volume visualization," *Eurographics Conference on Visualization (EuroVis)*, 2014.

[5] J. Xiang, F. Deng, and T. Ning, "The research of the hybrid volume rendering algorithm based on gpu technology." *Applied Mechanics & Materials*, no. 610, 2014.

[6] J. Liang, J. Gong, W. Li, and A. N. Ibrahim, "Visualizing 3d atmospheric data with spherical volume texture on virtual globes," *Computers & Geosciences*, vol. 68, pp. 81–91, 2014.

[7] P. Kumar and A. Agrawal, *Intelligent Interactive Technologies and Multimedia: Second International Conference, IITM 2013, Allahabad, India, March 9-11, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. CUDA Based Interactive Volume Rendering of 3D Medical Data, pp. 123–132. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37463-0_11

[8] M. Levoy, "Display of surfaces from volume data," *IEEE Comput. Graph. Appl.*, vol. 8, no. 3, pp. 29–37, May 1988. [Online]. Available: http://dx.doi.org/10.1109/38.511

[9] A. Weinlich, B. Keck, H. Scherl, M. Kowarschik, and J. Hornegger, "Comparison of high-speed ray casting on gpu using cuda and opengl," in *Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, vol. 1, 2008, pp. 25–30.

[10] L. Maršálek, A. Hauber, and P. Slusallek, "High-speed volume ray casting with cuda," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE, 2008, pp. 185–185.

[11] Y. Zhao, X. Cui, and Y. Cheng, "High-performance and real-time volume rendering in cuda," in *2009 2nd International Conference on Biomedical Engineering and Informatics*, Oct 2009, pp. 1–4.

[12] E. W. Bethel and M. Howison, "Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning," *International Jour-*

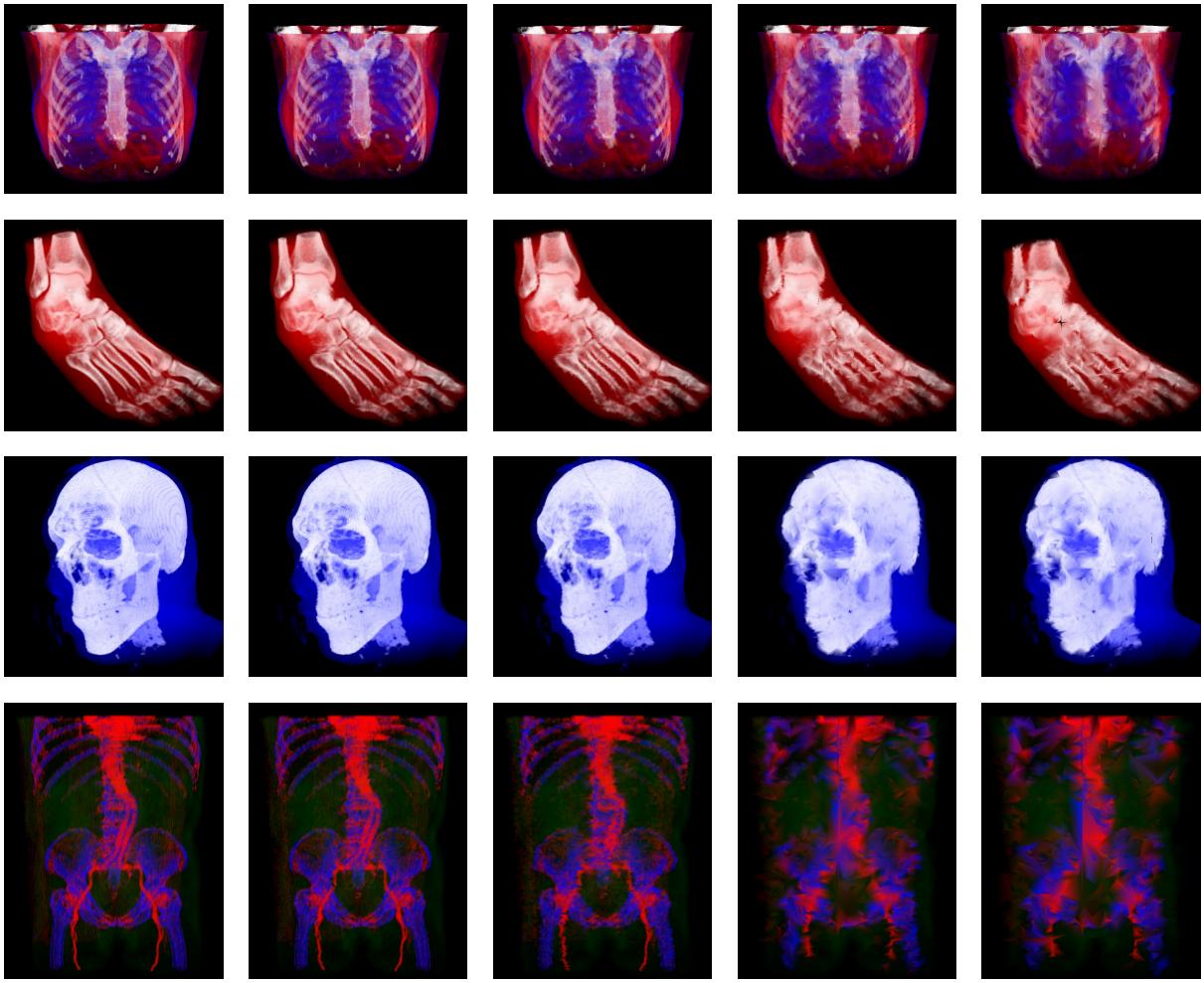


Fig. 11: Images rendered by VoS^{M*}. Isosurfaces decimated with the following decimation factors: 0% (first column), 25% (second column), 50% (third column), 75% (fourth column) and 90% (fifth column). From top to bottom: datasets of human chest, foot, head and stented abdominal aorta.

nal of High Performance Computing Applications, p. 1094342012440466, 2012.

[13] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78964.78965

[14] D. Tost, A. Puig, and I. Navazo, "Visualization of mixed scenes based on volume and surface," in *Proc. European Workshop on Rendering*, 1993, pp. 281–294.

[15] R. Westermann and B. Sevenich, "Accelerated volume ray-casting using texture mapping," in *Proceedings of the Conference on Visualization '01*, ser. VIS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 271–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=601671.601713

[16] B. Chen, A. Kaufman, and Q. Tang, *Volume Graphics 2001: Proceedings of the Joint IEEE TCVG and Eurographics Workshop in Stony Brook, New York, USA, June 21–22, 2001*. Vienna: Springer Vienna, 2001, ch. Image-Based Rendering of Surfaces from Volume Data, pp. 279–295. [Online]. Available: http://dx.doi.org/10.1007/978-3-7091-6756-4_19

[17] R. Marroquim, A. Maximo, R. Farias, and C. Esperança, "Volume and isosurface rendering with gpu-accelerated cell projection," in *Computer Graphics Forum*, vol. 27, no. 1. Wiley Online Library, 2008, pp. 24–35.

[18] A. Maximo, R. Marroquim, and R. Farias, "Hardware-assisted projected tetrahedra," in *Computer Graphics Forum*, vol. 29, no. 3. Wiley Online Library, 2010, pp. 903–912.

[19] NVIDIA. (2015) CUDA C Best Practices Guide. NVIDIA. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/