Snapping a Cursor on Volume Data

Wu, Shin-Ting and José Elías Yauri Vidalón
School of Electrical and Computer Engineering
University of Campinas
Campinas, Brazil
www.dca.fee.unicamp.br/∼{ting,elias}

Lionis de Souza Watanabe
Hospital de Clínicas
University of Campinas
Campinas, Brazil
lionis@hc.unicamp.br

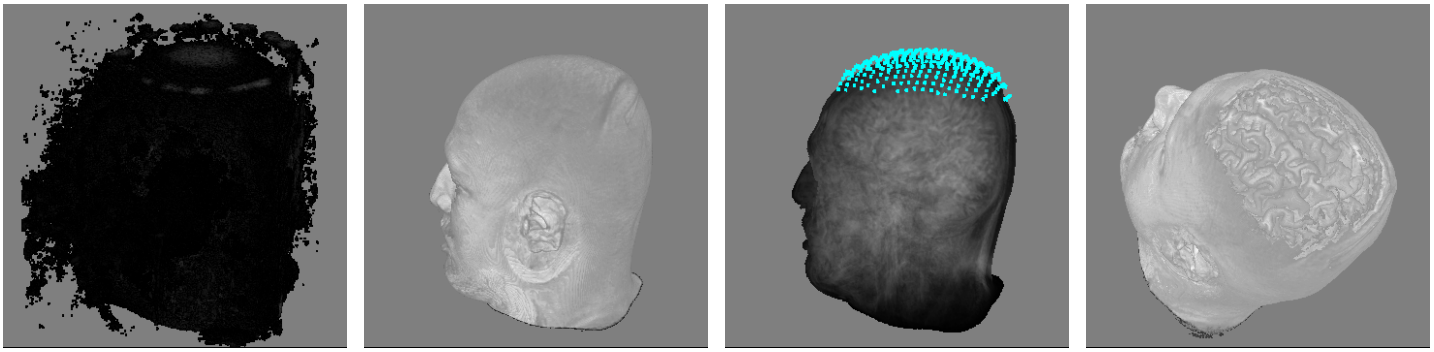


Fig. 1. Interactions on a raw data set (leftmost): after noise removal (left), painting on data with reduced opacity (right), and cropping (rightmost).

***Abstract*—In this paper we explore the architecture of GPU-based volume ray-casting to control the cursor movements on surfaces of volume data only on the basis of their sampled scalar values. This method does not rely on building a tight proxy geometry nor computing local iso-surface parametrization. In this setting only a few instructions should be included in a ray-casting shader: at every fragment the depth of the closest non-transparent voxel along each viewing ray is calculated and stored in the depth buffer. Its application in the manipulation of 3D medical images has, however, a broad spectrum. Two of them are presented. The proposed technique is very simple and fast, yet produces very nice and intuitive visual feedbacks.**

***Keywords*-3D Interactions; Ray-casting; Volume visualization; 3D medical images.**

I. Introduction

Volume visualization is recognized as one important form of displaying and exploring the internal structure of a 3D-array of sample data, such as 3D medical images acquired by X-ray computed tomography (CT) or magnetic resonance imaging (MRI) scanners. Researchers have proposed different rendering algorithms, classifiable either as a direct volume rendering approach [1] or an iso-surface extraction one [2]. Various types of transfer function specification methods have been proposed to highlight regions of interest and filter out irrelevant details [1]. More recent investigations have been focused on the multi-modal volume visualization [3], [4], [5], [6]. This may provide a physician more precise spatial relationships among the vascular structures, nerve fibers, physiologic function and anatomic structure, when the features of interest are "colored" distinguishably with help of appropriate multi-dimensional transfer function.

To address the lack of a robust 3D image segmentation algorithm, interventions of an expert is highly desirable to discriminate features of interest in a volume data set. One of the main obstacles in integrating a user in a segmentation process is, however, the difficulty of specifying those features in a spatial reference. Current applications are limited predominantly to 3D rigid transformations and 3D clipping algorithms, such as axis-aligned cuts [7], cuts at oblique angles [8] and in pre-specified geometry [9]. None of those tasks are fallen in the category of "tangible" interactions, in which the cursor of an input device should be *snapped* to the visible surface of the volume data. Unfortunately, the known tangible interactions, such as measuring the extension of the features of interest or curvilinear reformatting on the area of interest, are still limited to 2D interactions.

Mouse snapping can be seen as an special mouse picking mechanism that determines the current 3D positions of the cursor on a visible surface. Picking is the most commonly used intuitive operation to interact with 3D scenes in a variety of 3D graphics applications. It consists of selecting a geometric element pointed by a 2D or 3D cursor [10]. For a 2D cursor, a traditional implementation of picking consists in mapping the 2D mouse position in the screen space onto a set of objects along the *mouse ray* in the 3D scene space. The nearest object to the viewer that intersects the mouse ray is selected. To extend this to surface snapping, instead of object, the nearest 3D intersection point is returned. Performing this continuously

in time will produce the perception that the cursor is snapped to the surface [11].

The basic operation of a tangible interaction is, therefore, the determination of the x, y and z coordinates of where a user intends to touch in the 3D scene space, from the 2D mouse position in the screen space where s/he actually clicks. The crucial difference of our work from previous ones is that in the volume visualization only the shape of the data domain is known, not the shape of the object contained in the data [1].

Contributions: This paper aims at an essential task for tangible interactions with volume data, even when the shape of the object of interest is not analytically given. Our proposal, based on "what you see is what you snap" paradigm, is a hybrid of two well-known techniques: ray-casting picking [12] and depth buffering [13]. It explores the feature of the former that delivers correct world coordinates by just intersecting the mouse ray against the volume elements, while it takes advantage of the latter that avoids per pixel single-pass ray-casting whenever the mouse is moved on the 3D rendition of the volume data. We show that our proposal only requires slight adjustments in the existing ray-casting volume rendering techniques to make them appropriate for interactive visualization. The key to our solution is to store in the depth buffer the z value of the closest visible voxel along the viewing ray in the same way as the fixed-function graphics pipeline stores, so that the traditional depth buffering approach may be employed for getting the actual 3D location that the user intends to click on. We also apply our proposed technique for implementing two tangible interactions with 3D medical image data sets: painting and measuring.

A. Related work

There are two approaches to map back a 2D screen point to a 3D scene point. One method, available in the D3DX extension library of the industry standard API Direct3D [14], is to generate a ray using the mouse's location and then intersect it with the world geometry, finding the object nearest to the viewer. It works by locking the model's vertex and index buffer, computing a ray-triangle intersection test with each face and sorting the intersection points according to the distance to the viewer. Alternatively we can determine the actual 3-D location that the user has clicked on by sampling the depth buffer and performing an inverse transformation, as provided both in D3DX and in OpenGL [15]. As for volume rendering the shape of the object contained in the volume data is not given, in both approaches the application must first estimate its surface and then perform the object space intersection test with the resulting geometry. We propose in this work a way to circumvent this pre-processing by exploring the volume ray-casting features.

To snap a cursor to the surface of 3D models whose geometry is arbitrarily deformed by a programmable hardware fragment and vertex processor, Batagelo and Wu introduced the idea of snapping a 3D cursor to the visualized surface fragment [16]. They extended the idea to volume data sets by snapping the cursor to its iso-surfaces [17]. Bürger *et al.* presented a similar interaction paradigm [18]. They introduced the idea of *surface particles* to map annotations onto the iso-surfaces at sub-voxel accuracy. Actually, the way that they move a particle on the surface had been previously described by Wu *et al.* in [11]. With such improved interactive tools one may easily specify any visible region to be edited. In addition, Bürger *et al.* showed that one may use the same annotation grid to create a shape aligned windowed-cutaway section. The procedures provide very nice interactivity. Nevertheless, for computing 3D position where the cursor is snapped it is necessary to shoot for each mouse position a pick ray into the volume data in order to determine if and which voxel the user has clicked on the volume data. In this work we show that we may combine the volume ray-casting and the volume ray picking into a single-pass processing, since the depth buffering is supported.

Chen *et al.* presented a mask-based interaction tool for peeling, cutting and pasting [19]. Through a sequence of input points provided by the user, their system generates a mask covering the area of interest. The authors have remarked that the mask only works appropriately if the selected surface contains a smooth change of gradients. This is because the procedure relies heavily on the estimated gradients. When the volume data are very noisy, the geometry of their iso-surfaces is rough, compromising gradient smoothness. Although a smoothing filter may attenuate the roughness, the original data values may be prohibitively changed. Our technique does not alter the original scalar values. We explore the interactivity letting expert remove the noise through two interaction modes: threshold level setting and paintbrush erasing.

B. Technique overview

Provided that there is a depth buffer in the graphics pipeline, Benstead presents in [13] a C source code that converts screen 2D coordinates to the 3D coordinates where a user in fact clicked on the object space. We further consider that the 2D projection of the 3D data set is rendered in a single-pass GPU volume ray-casting. The volume data are stored as a 3D texture in the texture memory and is mapped to a proxy geometry which is actually rendered. Fragments are generated after rasterization and processed by a fragment shader. Our proposal consists in simply storing the depth of the nearest non-transparent voxel to the viewer in the depth buffer when a *light ray* is traversed in the shader loop. After then, to mitigate performance penalty we make a copy of the information stored in the depth buffer. This information is used by an event handler to interactively control the mouse's spatial position in the CPU while the 3D rendition is not changed. Visual feedbacks are either sent to programmable graphics pipeline or directly to the framebuffer. The process is schematized in Fig. 2. The modules of a programmable graphics pipeline that are involved in processing are highlighted in thicker, red line. Also observe the integration of the user's actions in the process.

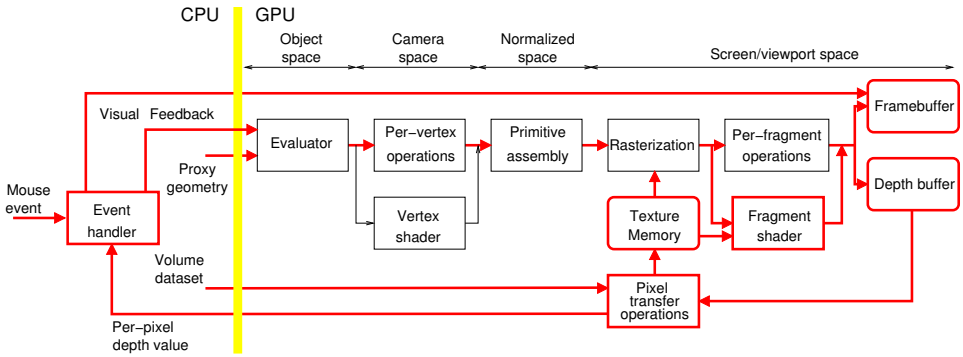


Fig. 2. Data-flow of our proposed snapping algorithm.

II. Technical Background

To take full advantage of the depth buffering in the fixed-function pipeline, we only need to calculate in the fragment shader the z-values of the closest non-transparent voxel to the viewer in the same way as that pipeline would compute. We have worked on how to compute these z-values.

To be self-contained, it is convenient to give a brief description of the components involved and their interfaces. Those details are particularly relevant here since our proposal explores the features presented in the existing technologies and fills the gaps between them. The components are: the basic structure of ray-casting, the transformations from the object space to the screen space in the fixed-function pipeline, and a mouse picking method based on depth buffering in the fixed-function pipeline.

A. Volume rendering with ray-casting

The basic goal of volume rendering is to estimate per pixel the light intensity that reaches the viewer after traversing the volume data along the light ray in the object space. One pragmatic approach is to resample the volume data at regular intervals along the ray, as shown in Fig. 3. By means of an appropriate transfer function $\mathcal{TF}$, optical properties, namely color C_i and opacity α_i, are assigned to each sample i. These optical properties are recurrently composited in the same order as the ray traversal, usually in the front-to-back order (i=0 to i=n), to provide a final pixel intensity C_n and opacity α_n [1]:

$$\begin{aligned} C_i &= C_{i-1} + (1-\alpha_{i-1})C_i \\ \alpha_i &= \alpha_{i-1} + (1-\alpha_{i-1})\alpha_i. \end{aligned}$$

The initial values are the values of the first sample along the ray $C_0 = C_0$ and $\alpha_0 = \alpha_0$. The results are clamped to $[0, 1.]$. Engel *et al.* describe how this procedure can be implemented on GPU architectures via fragment shader functionality [1]. Two relevant features of their presented algorithms have been explored in our work.

First, observe that when a sample has $\alpha_i = 0$, its color is effectively discarded from the composition and, consequently, not "visible". For example, we consider in Fig. 3 that the samples with $\alpha_i = 0$ are drawn as circles with dotted lines.

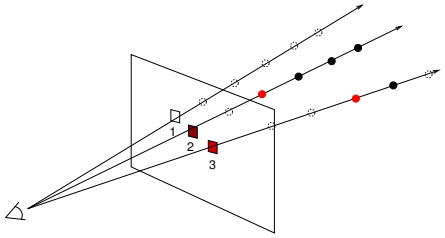
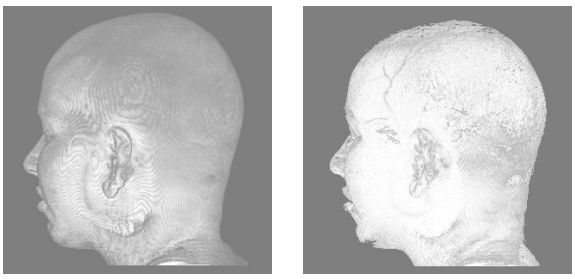


Fig. 3. Ray-casting.

Applying the composition procedure, we get "no color" for the pixel 1 and "opaque" final colors to the pixels 2 and 3. This correspondence between opacity and visibility is an important ingredient for the "what you see is what you snap" interaction paradigm. Through an appropriate transfer function from the scalar to the opacity values, it is possible to control the contribution of each voxel to the final 3D rendition.

Second, the initial value C_0 does not have to be the first voxel with scalar value greater than zero along the viewing ray. It could be a color corresponding to a threshold density level d_{th} chosen by the user, i.e. $C_0 = \mathcal{TF}(d_{th})$, such that the composition only starts with the voxels that meet it. In the case of medical volume data, most of organs have their own scalar levels d and they may be used to selectively enable or disable distinguishable objects of interest contained in the volume data. Fig. 4 illustrates the visual effects of two distinct values of d_{th} in the same volume data.



(a) (b)

Fig. 4. Ray-casting with (a) $d_{th} = 1375$ and (b) $d_{th} = 2803$.

B. Projection transformation

OpenGL graphics pipeline breaks the transformation from the object space to the screen space down into several space transformations. In Fig. 2 two intermediary spaces along the fixed-function graphics pipeline are distinguished. We denote the matrix transformation from the camera space to the normalized (device) space as $\mathcal{P}$. It is called *projection matrix*. In the normalized space the range of coordinates is [-1.0,1.0]. The transformation from the coordinates (n_x, n_y, n_z) of the normalized space to the coordinates (s_x, s_y) of the screen space, with dimensions $w \times h$ and the leftmost corner at (x_0, y_0), is given by the transformation [15]:

$$s = \begin{bmatrix} s_x \\ s_y \\ s_z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} + x_0 \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} + y_0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \mathcal{V}n. \tag{1}$$

This transformation matrix is referred to as the *viewport matrix*. Observe that the per-fragment operations in the fixed-function pipeline are applied to the coordinates $(s_x, s_y, s_z, 1)$, where s_z is the depth value stored in the depth buffer of the fixed-function pipeline. This depth value is in the range $[0.0, 1.0]$, with 0.0 at the near clip plane and 1.0 at the far clip plane.

In addition, it is allowed to transform affinely each scene object from its own object space to the camera space. The composition of such transformations may be described by a *view matrix* $\mathcal{M}$. The use of $\mathcal{M}$ and $\mathcal{P}$ allows us to algebraically express the conversion of the coordinates $p = (x, y, z, 1)$ to the coordinates $(s_x, s_y, s_z, 1)$. We simply plug the result of the conversion $n = \mathcal{P}\mathcal{M}p$ into Eq. 1 and get

$$s = \mathcal{V}\mathcal{P}\mathcal{M}p$$

In the GLSL shading language, there are built-in uniform variables gl_ModelViewMatrix and gl_ProjectionMatrix permitting shaders to access, respectively, the state of matrices $\mathcal{M}$ and $\mathcal{P}$. Direct access to the composition $\mathcal{P}\mathcal{M}$ is also possible through the variable gl_ModelViewProjectionMatrix [20].

The knowledge of these transformation functions is helpful in the computation of the depth s_z in a fragment shader. We should, however, draw attention to the largest issue with performance: writing to gl_FragDepth prevents early-z culling. It is, therefore, best to reduce fragment depth updates to avoid overdraw.

C. 2D to 3D mouse coordinate conversion

In the fixed-function OpenGL graphics pipeline, if the depth buffering is enabled, the depth value s_z of each rendered pixel, normally the pixel that is closest to the viewer, is automatically stored in it. This depth value is determined based on the view matrix $\mathcal{M}$, the projection matrix $\mathcal{P}$, and the viewport matrix $\mathcal{V}$, as explained in Section II-B. In addition, there is a utility library providing a function that converts the coordinates in the object space to the coordinates in the screen space from those matrices, or vice-versa. They are, respectively, glProject and glUnProject. In this section we summarize the main steps of the procedure.

To initialize the conversion procedure, the depth buffer should be enabled and the mouse motion callback specified. Then, whenever the mouse moves, its position is captured by an event handler and the corresponding callback issued (Fig. 2). In this callback, the function glReadPixels is used to get the s_z depth of the surface rendered in the pixel (s_x, s_y). After retrieving the matrices $\mathcal{M}$, $\mathcal{P}$ and the viewport parameters with the function glGet, the function glUnProject is applied to map the coordinates (s_x, s_y, s_z) onto the coordinates (x, y, z) in the object space.

Provided that the depth buffer is correctly filled, this handling fits our proposal like a glove.

III. Our proposal

In this section we present a simple, yet effective, way to snap a cursor on the visible surface of a volume data set. We aim at proposing a procedure that maximally reuses the existing tools presented in Section II and requires minimal changes in the existing ray-casting based volume rendering techniques available as a fragment shader.

A. Problems

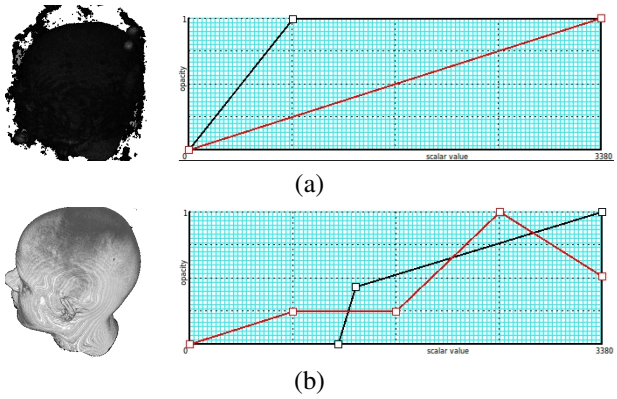
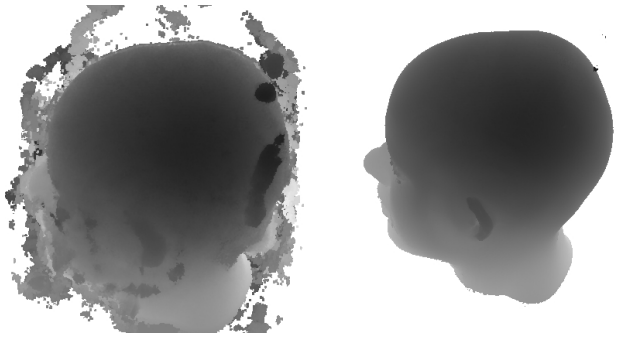


Fig. 5. Two images resulting from distinct transfer functions (scalar value $\times$ {opacity,grayscale}).

As already stated, the basic problem that should be solved is how to efficiently calculate the depth value for the nearest opaque sample to the viewer, such as the red circles in Fig. 3. The opacity of each sample is, nevertheless, strongly dependent on the underlying transfer functions, as illustrates Fig. 5 in which the red and black lines depict, respectively, the color and the opacity transfer functions. Observe that in Fig. 5.(a) all samples with scalar values greater than zero have their light information accumulated in the pixel's color, while in Fig. 5.(b) only samples with values greater than a prespecified threshold, in this case 1317, enter into the light composition.

Fig. 6.(a) and Fig. 6.(b) show, respectively, the corresponding depth maps of Fig. 5.(a) and Fig. 5.(b). Note that the darker the shade is, the closer from the viewer is the sample. If the purpose is interacting with the head's scalp, the latter transfer



(a) (b)

Fig. 6. Depth buffer of (a) Fig. 5.(a) and (b) Fig. 5.(b).

function is much more appropriate as all noisy data have been removed. On the other side, we may miss relevant internal structures that possess scalar values less than a threshold, compromising the rendering quality of the regions of interest. This is the second problem that we should solve to ensure that our proposal is useful in practice.

B. Depth computation

For meeting the "what you see is what you snap" requirement, we adopt the front-to-back ray traversal order and take as the visible sample along the mouse ray the first ray position p that is in a non-transparent voxel. Obtaining p we should calculate its depth value in the range [0.0,1.0] and store it in the depth buffer. For reuse the procedure presented in Section II-C, this value should be consistent with the one that the fixed-function graphics pipeline computes.

Since in the fragment shader all the sample position and direction vectors are described with respect to the object space, we use gl_ModelViewProjectionMatrix to determine the depth of p with respect to the normalized space, as shown in Section II-B

$$n = (gl\_ModelViewProjectionMatrix)p.$$

Then, we apply Eq. 1 to find the depth value in the screen space

$$s_z = \frac{1}{2}n_z + \frac{1}{2}.$$

To alleviate the performance drop the drawing mode GL_FRONT and the face culling are applied in the 3D rendition of the proxy geometry.

C. 3D Eraser tool

For circumventing the noisy samples that hinder desirable direct manipulations, we may in principle either provide an appropriate transfer function or employ an adequate 3D image segmentation function. However, the existing solutions fall far short of covering a large variety of situations and we end up adopting a trial-and-error paradigm for dealing with unsolvable situations. In this work, we propose manual removal of the noisy data by an expert. The interactivity of our proposed snapping algorithm and the rendering quality is a good combination for an intuitive interface. The user can erase the noises to transparency with a few interactions.

Similar to the paintbrush eraser in the Adobe® Photoshop [21], the user can erase the sample data by simply painting them in transparency. The difference is, in the place of a 2D space, our proposed eraser tool works in the 3D domain. In order to preserve the original volume data, we set up a *control volume* that has the same spatial resolution as the original one and its voxel values are used to control the modulation of the original volume data. In the fragment shader, both the original volume data and the control volume data are used in combination for deciding the light contribution of each data sample. When a sample is tagged as transparent in the control volume, it is simply skipped and the feedback to the user is that the sample is no longer visible. If there are more noisy data behind the erased ones, they are going to become visible and further interactions are necessary to "remove" them completely.

D. Thresholding

The erasing interaction mode is, however, very stressful and time-consuming. It should preferably be applied only as an option for retouching a few unwanted samples. To remove the majority of the noisy data that obstruct the visibility of the voxels that the user wants to see without altering the opacity of the voxels behind those wanted voxels, we propose to use "transfer functions" that also consider the threshold density level d_{th}, explained in Section II-A, for assigning opacity and color to the range of the scalar values. We call this value the *noise threshold*. In our "transfer functions" a scalar value $d < d_{th}$ may have two correspondences. If it has $d < d_{th}$ and lies between the observer and the first voxel with d_{th} from the observer to the volume data, its opacity and color are set to zero; otherwise we use the pre-specified transfer functions to assign the optical properties.

Fig. 1.(b) illustrates the result of the actions of our proposed eraser tool on the volume data depicted in Fig. 1.(a). The noise threshold was set at 1600 and some remaining noisy samples have been painted in transparency.

IV. Implementation

Based on our proposal, any single-pass GPU ray-casting should be adaptable to make it able to perform mouse picking at interactive rate. To validate that our procedure is easily integrable into an existing ray-caster, we chose an implementation of the texture-based ray-casting architecture developed by Stegmaier *et al.* [22]. The ray-caster was implemented in C++ using GLSL. In this section we present the modifications that we have included.

Besides the original volume data, we need to store the control volume as a 3D texture. According to the tag set in its entry, the shader either reads the optical properties $dstColor$ of the data sample (x, y, z) or skip it. To accomplish it we added the following if-statement in the fragment shader

$$tag \Leftarrow ControlVolume[x][y][z]$$

if tag is transparent **then**
$dstColor \Leftarrow (0, 0, 0, 0)$
else
$idx \Leftarrow OriginalVolume[x][y][z]$
$dstColor \Leftarrow TransferFunction[idx]$
end if

In the fragment shader loop that traverses a light ray, we need to look for the first non-transparent sample with $d \geq d_{th}$ among the traversed points P and determine its depth. Another if-statement was inserted for carrying out this task

if P is the first non-transparent sample and $d(P) \geq d_{th}$ **then**
$n \Leftarrow gl\_ModelviewProjectionMatrix \cdot P$
$depth \Leftarrow 0.5 * n.z + 0.5$
end if

And before leaving the shader, the fragment's depth value must be updated

gl_FragDepth = $depth$

Done these adjustments, the depth buffer is filled with the depth values of all visible data samples whenever its proxy geometry is rendered. The implemented fragment shaders are available in [23].

V. Experiments

We validated our proposed snapping technique through a series of experiments on a desktop Intel Core2 Duo E7500 2.936 GHz CPU with a NVIDIA GeForce GT240 GPU. The experiments have been performed on the 3D medical images. The data were acquired either by an Elscint 2T Prestige MRI Scanner or by a RM 3T Philips Intera-Achieva Scanner at our university hospital.

First, the evaluation was done to assess the performance drop. We measured the variations in the time spent to carry out 3D rendition, with and without the computation of depth map, of a set of 240×240×180 volume data for different output image resolutions, from 64×64 to 1362×1362. Then, we measured the time spent by glReadPixels to read the depth map back from the depth buffer for these output resolutions.

Second, we checked how our proposal is sensible to the variations of the opacity transfer function. We would like to know whether our proposal meets the "what you see is what you snap" requirement, i.e. whether the depth buffer is correctly filled, even when the opacity of the samples are very low. We kept the noise threshold constant in the value 423 and varied the opacity of the volume data. We observed that, except when the opacity is zero (totally transparent), the depth map is opacity insensitive. Fig. 7 illustrates five renditions with distinct opacities and their corresponding depth map. In particular, the color clamp effect to 1.0 is shown in Fig. 7.(d).

Third, we compared the visual effects of the noise threshold parameter and the opacity transfer function. As we claimed, the adjustments through the noise threshold parameter is local, making selectively some samples transparent, while the transfer functions affect globally the optical properties of the

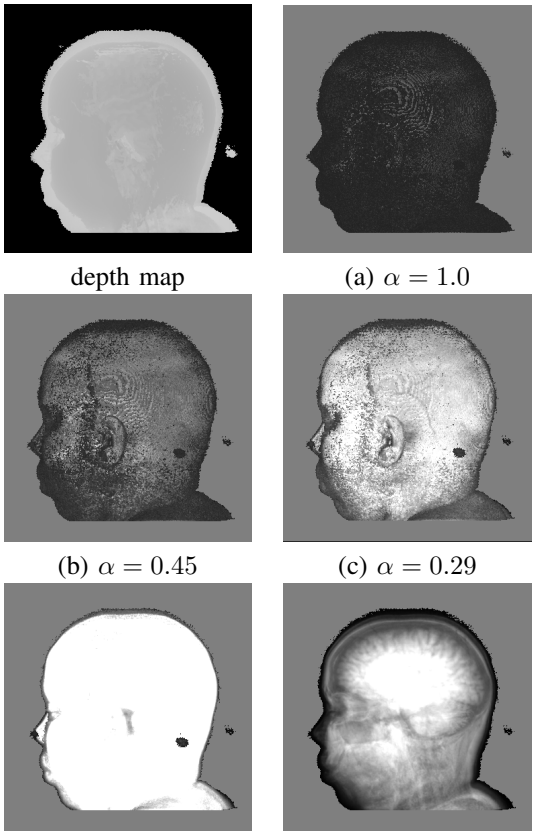


Fig. 7. Distinct opacity transfer functions, from the highest (a) to the lowest opacity (e), lead to the same depth map.

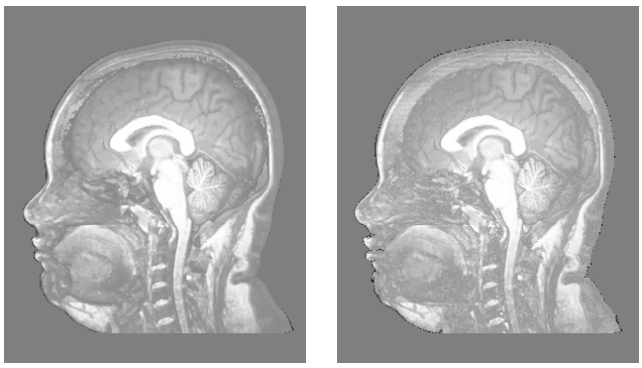


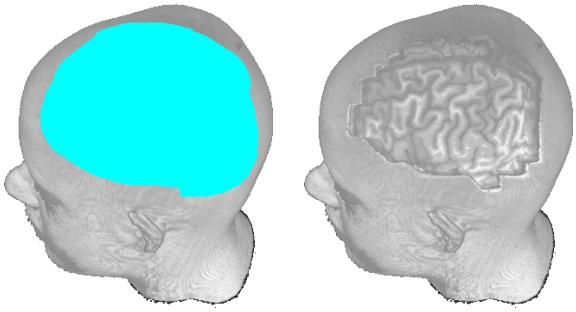
Fig. 8. Noise removal with (a) noise threshold and (b) opacity transfer function.

samples. Observe the subtle differences in the sharpness and in the structure between Fig. 8.(a) and Fig. 8.(b). In the former the noisy data have been removed by setting the noise threshold in 1056 and in the latter the opacity of the voxels with scalar values in the range $[0, 1056]$ has been assigned zero.

Finally, we show how to apply our proposed technique in implementing two tangible interactions with a volume data set: painting and measuring.

A. Painting

Painting is mostly employed in volume editing, affecting primarily in the optical properties of the picked data samples. It provides a nice visual feedback that may guide the user in a variety of tasks, such as selection and segmentation. It works just like counterparts its eraser tool explained in Section III-C. As the mouse is moved on the visible data samples, their spatial locations are continuously computed and the corresponding voxels in the control volume are accordingly tagged. These tags are used to switch the transfer functions and the optical properties of the painted samples are immediately changed to the pre-specified ones. More to highlight some parts of the object of interest, the painting tool may be employed in specifying the samples of an initial geometry, as illustrates Fig. 1.(c), for further processing. Fig. 1.(d) and Fig. 9 present the application of painted samples in cropping away the shell of a volume data set in order to expose its internal structures [24].



(a) (b)

Fig. 9. Cropping: (a) painting and (b) removal of the painted region.

B. Measuring

Measuring is an important tool for evaluating both the size and the extension of some features. It follows the same interaction principle as the painting. First the user selects a sequence of points $\{p_0, p_1, p_2, \ldots, p_n\}$ on the curve of interest by clicking on them. We use our proposed method to find the coordinates $(x_i, y_i, z_i, 1)$ of those points in the object space. For providing visual feedback, the corresponding voxels in the control volume are tagged in accordance with the previously attributed tags. In addition, the distance between the samples are estimated, so that when the last sample of a sequence is inserted, the length L is computed from

$$L = \sum_{i=1}^{n} \|(k_x(x_i - x_{i-1}), k_y(y_i - y_{i-1}), k_z(z_i - z_{i-1}))\|,$$

where k_x, k_y and k_z are the voxel spacing values in the x-, y-, and z-direction respectively. Fig. 10.(a) illustrates the visual feedback of a sequence of points from the interaction view and Fig. 10.(b) show another view to emphasize how the points are "stitched" to the visible surface. With our 3D measurement algorithm, the depth and the extent of the cropped region were computed. It is 19mm in depth, 90mm in width and 108mm in length.

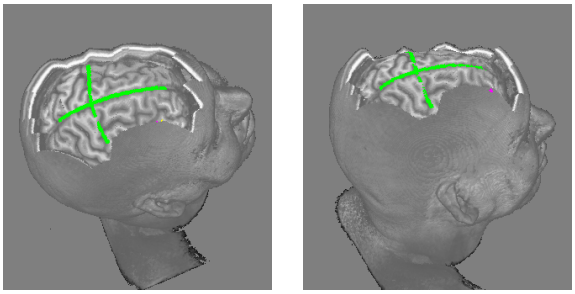


Fig. 10. Measuring the length of user specified curves.

VI. Results and Discussion

We have evaluated our proposed method in view of the following aspects:

Performance: As only two if-statements and one assignment are included in the classical ray-caster to make it suitable for interactions, time performance degradation is expected to be almost imperceptible in view of interactivity. In fact, the volume data can be, at interactive rate, moved around the screen and manipulated by controlling a cursor with a mouse. Nevertheless, our expectation has not been corroborated by the performance experiments described in Section V. The performance presents larger time variations with the output resolution when the depth buffer is enabled. Table I summarizes the frame rates for different rendering modes: without depth buffer, with depth buffer, and with depth buffer and optimized rendering mode explained in Section III-B. Observe in the fourth column that till 400^2 the degradation is imperceptible, but when the output resolution further increases, the performance drops almost linearly. This behavior may be due to the number of parallel cores and video memory available in the graphics card.

TABLE I
Performances results in fps.

Resolution	No Depth	Depth	Depth+Optim.	Degradation
128^2	62	62	61	0
400^2	60	32	60	0
512^2	60	21	57	-3
700^2	60	12	36	-24
900^2	59	7	22	-37
1362^2	58	3	10	-48

With respect to the performance of reading data back from the depth buffer, the time variations only depend on the output resolution. For the image resolutions presented in Table I the times spent are, respectively, about 0.06ms, 0.47ms, 0.78ms, 4.22ms and 9.69ms.

Visual quality: To assess the quality of the outcomes, we asked neurologists and neurosurgeons working at our neuroimaging laboratory to manipulate and explore the volume data with the interaction tools we developed. Mostly two or three attempts suffice to identify serious injuries. No more than ten attempts were necessary for a specialist in dysplasia diagnosis to discover subtle abnormal patterns. One motive given by the doctors for their rapid finding is that the interface of

our prototype is much faster and easier to use than the systems currently installed at our neuroimaging laboratory [25], [26], [27]. It is worth commenting that one reason for presenting 3D rendition in grayscale is that the physicians are used to this art of display.

Domain coverage: Although we have used ray-casting based volume rendering approach in this work, other rendering methods may be used. The key ingredient is to find out the first visible element in the view direction that is nearest to the viewer and to store its depth, given in the screen space, in the depth buffer.

Limitations: The main limitation of our proposed technique is that it is not suitable for noisy data. Nevertheless, we devised an intuitive way for the user to erase these noises by combining the flexibility of a transfer function editor and the interactivity of our mouse picking method. The erasing tool has not been exhaustively validated yet. This validation should be conducted shortly. As a mid-term goal we would like to search for transfer functions more appropriate for 3D medical images.

VII. Conclusion

We presented a basic tool to interact with volume data even when their shape is not describable analytically. We showed that our tool is particularly simple to be integrated in the existing volume rendering implementation. Only commands to detect the visible samples and to compute their depth in the screen space are necessary. Despite its simplicity in implementation, our proposed technique has potential application in a large spectrum of 3D interaction tasks with volume data. Two of them have been detailed in this paper. As further work we plan to develop a pre-surgical planning tool for removal of cortical lesions on top of our proposal.

Acknowledgment

The authors would like to thank the colleagues at their university hospital, in particular Dr. Clarissa L. Yasuda and Prof. Dr. Fernando Cendes, for providing medical images and valuable feedbacks. This work has been supported by CAPES (Coordination for the Improvement of Higher Level Personnel), FAPESP (The State of São Paulo Research Foundation) under grant no. 2011/02351-0.

References

[1] M. Hadwiger, J. M. Kniss, C. Rezk-Salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[2] A. C. Telea, *Data Visualization: Principle and Practice*. AK Peters, Ltd, 2007.

[3] D. Valentino, J. Mazziotta, and H. Huang, "Volume rendering of multimodal images: application to mri and pet imaging of the human brain," *Medical Imaging, IEEE Transactions on*, vol. 10, no. 4, pp. 554–562, Dec. 1991.

[4] R. J. Frank, H. Damasio, and T. J. Grabowski, "Brainvox: An interactive, multimodal visualization and analysis system for neuroanatomical imaging," *NeuroImage*, vol. 5, no. 1, pp. 13–30, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/B6WNP-45KKTNX-17/2/43e0a7bff46654ab121941deb138bd38

[5] T. Ropinski, M. Specht, J. Meyer-Spradow, K. H. Hinrichs, and B. Preim, "Surface glyphs for visualizing multimodal volume data," in *Proceedings of the 12th International Fall Workshop on Vision, Modeling, and Visualization (VMV07)*, nov 2007, pp. 3–12. [Online]. Available: http://viscg.uni-muenster.de/publications/2007/RSMHP07

[6] C. Rieder, M. Schwier, H. K. Hahn, and H.-O. Peitgen, "High-Quality Multimodal Volume Visualization of Intracerebral Pathological Tissue," C. Botha, G. Kindlmann, W. Niessen, and B. Preim, Eds. Delft, The Netherlands: Eurographics Association, 2008, pp. 167–176. [Online]. Available: http://www.eg.org/EG/DL/WS/VCBM/VCBM08/167-176.pdf

[7] C. Roden and M. Brett, "Stereotaxic display of brain lesions," *Behavioural Neurology*, vol. 12, pp. 191–200, 2000.

[8] C. Rezk-Salama, K. Engel, and F. V. Higuera, "The OpenQVis Project," accessed in April 2011. [Online]. Available: http://openqvis.sourceforge.net/

[9] D. Weiskopf, K. Engel, and T. Ertl, "Interactive clipping techniques for texture-based volume visualization and volume shading," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 298–312, 2003.

[10] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Reading, MA: Addison-Wesley Publishing Co., 1990.

[11] S.-T. Wu, M. Abrantes, D. Tost, and H. C. Batagelo, "Picking and snapping for 3d input devices," *Brazilian Symposium on Computer Graphics and Image Processing*, pp. 140–147, 2003.

[12] T. O. S. Project, "Mouse picking demystified," http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/MousePicking, accessed in April 2011.

[13] L. Benstead, "Using gluunproject," http://nehe.gamedev.net/data/articles/article.asp?article=13, accessed in April 2011.

[14] *DirectX 9.0 Programmer's Reference*, Microsoft Corporation, October 2004.

[15] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[16] H. C. Batagelo and S.-T. Wu, "What you see is what you snap: snapping to geometry deformed on the GPU," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics, SI3D 2005*, Washington, DC, USA, 2005, pp. 81–86.

[17] H. C. Batagelo and S.-T. Wu, "A framework for GPU-based application-independent 3D interactions," *The Visual Computer*, vol. 24, no. 12, pp. 1003–1012, 2008.

[18] K. Bürger, J. Krüger, and R. Westermann, "Direct volume editing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1388–1395, 2008.

[19] H.-L. J. Chen, F. F. Samavati, and M. C. Sousa, "GPU-based point radiation for interactive volume sculpting and segmentation," *The Visual Computer*, vol. 24, no. 7, pp. 689–698, 2008.

[20] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, Jan. 2006.

[21] B. Brundage, *Photoshop Elements 9: The Missing Manual*, 1st ed. Pogue Press, 2010.

[22] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting," in *Volume Graphics*, 2005, pp. 187–195.

[23] W. S. Ting, J. E. Y. Vidalón, and L. de Souza Watanabe. [Online]. Available: http://www.dca.fee.unicamp.br/projects/mtk/wu/vmtk2.html

[24] S.-T. Wu, C. L. Yasuda, and F. Cendes, "Interactive curvilinear reformatting in native space," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, no. PrePrints, 2011.

[25] R. R. Inc., *BrainSight – User Manual, Version 1.7*, Accessed in February 2010. [Online]. Available: http://www.icts.uci.edu/neuroimaging/BransightTMS.pdf

[26] F. Bergo and A. X. Falco, "Fast and automatic curvilinear reformatting of MR images of the brain for diagnosis of dysplastic lesions," in *2006 International Symposium on Biomedical Imaging*, Arlington, Virginia, USA, 2006, pp. 486–489.

[27] G. Flandin and K. J. Friston, "SPM2 – Statistical Parametric Mapping," accessed in July 2011. [Online]. Available: http://www.fil.ion.ucl.ac.uk/spm/software/spm2/