International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:3, 2010

GPU-Based Volume Rendering for Medical Imagery

Hadjira Bentoumi, Pascal Gautron, and Kadi Bouatouch, *Member, IEEE,*

Abstract**—** We present a method for fast volume rendering using graphics hardware (GPU). To our knowledge, it is the first implementation on the GPU. Based on the Shear-Warp algorithm, our GPU-based method provides real-time frame rates and outperforms the CPU-based implementation. When the number of slices is not sufficient, we add in-between slices computed by interpolation. This improves then the quality of the rendered images. We have also implemented the ray marching algorithm on the GPU. The results generated by the three algorithms (CPU-based and GPU-based Shear-Warp, GPU-based Ray Marching) for two test models has proved that the ray marching algorithm outperforms the shear-warp methods in terms of speed up and image quality.

Keywords**—** Volume rendering, graphics processors

I. Introduction

Direct volume rendering methods [1] generate images of 3D volumetric data sets without knowing any geometrical information about the objects captured, such as human organs, skull, etc. These methods make use of an optical model to assign optical properties (color and opacity) to the data. When rendering the data sets optical properties are accumulated along each viewing ray. The volumetric data set is represented by a uniform 3D array of samples. The optical properties are either specified by the data values directly or computed using a transfer function that is applied to the data.

Two kinds of direct volume rendering techniques have been proposed in the literature : CPU-based and GPU-based.

The CPU-based approaches do not make use of the GPU, so they are implemented entirely on the CPU. For acceleration purposes, these methods resort to hierarchical data structures such kD-trees [2] and octrees [1], [3]. These data structures are well suited to skipping empty regions. The most efficient CPU-based direct volume rendering method is the Shear-Warp algorithm [3].

As for the the GPU-based methods, they can be divided into two categories. The methods of the first category make use of 2D or 3D textures. Indeed, they consider the data (a discrete 3D scalar field) as a stack of 2D texture slices or as a single 3D texture [4], [5], [6], [7]. The common principle of the methods is the visualization of a high number of semi-transparent 2D slices (2D textures or textured proxy geometries) extracted from the 3D scalar field. The polygons corresponding to these slices are geometric primitives that have to be rendered. These primitives represent only a proxy geometry. The problems of slice-based volume rendering are: it is limited to rasterization, it has difficulties to make use of acceleration methods, it is inflexible.

Regarding the methods of the second category, they are based on ray casting (also called ray marching) [8], [9], [6]. The principle of ray casting consists in casting rays from the center of projection of the camera toward the volume data and compute the volume rendering integral along these rays. In contrast to slice-based rendering, ray casting is capable of addressing the problems encountered by slice-based volume rendering. One of the advantages of ray casting is that the cast rays are processed independently from each other. This allows for optimizations strategies such as: early termination, adaptive sampling and empty space skipping.

The objective of this paper is twofold. First, we propose a GPU-based implementation of the Shear-Warp algorithm [3]. To our knowledge, it is the first implementation on the GPU. Second, we compare it to a CPU-based Shear-Warp algorithm and to a GPU-based ray casting in terms of flexibility, frame rate and quality of the rendered images.

The rest of this paper is structured as follows. Section 2 outlines the Shear-Warp algorithm followed by a CPU-based implementation depicted in section 3. A GPU-based implementation of Shear-Warp is described in Section 4. Section 5 briefly presents our GPU-based ray casting method. A comparison of the three methods as well as other results are given in Section 6 before concluding in Section 7.

II. Shear-Warp Algorithm: An Overview

The Shear-Warp algorithm [3] relies on a factorization of the viewing transformation. This method can be split into two steps (Fig. 1). In the first step, the volume slices are *sheared* according to the viewing direction. This operation yields an *intermediate image*, containing a distorted image of the volume as seen from the viewpoint (Fig. 2(a)). This distortion is compensated by warping the intermediate image (Fig. 2(b)).

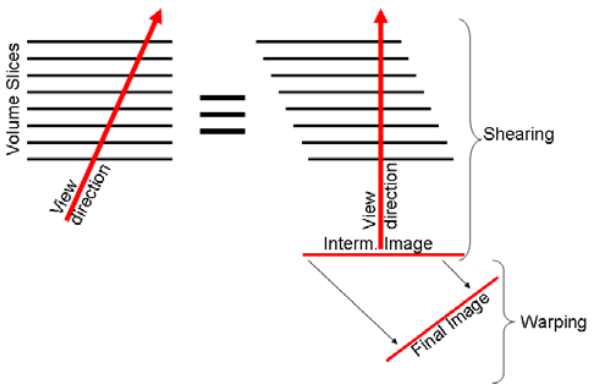


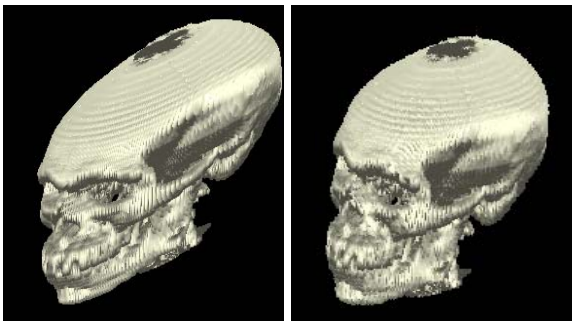
Fig. 1. The Shear-Warp algorithm. Rendering a sliced volume is replaced by shearing the volume slices according to a given viewing direction, yielding a distorted intermediate image. This image is then warped to obtain the final image of the volume as seen from this direction.

H. Bentoumi is with . e-mail: hbentoumi@wissal.dz
P. Gautron is with the IRISA/INRIA Rennes. e-mail: pgautron@irisa.fr
K. Bouatouch is with the IRISA/INRIA Rennes. e-mail: kadi@irisa.fr

For the sake of clarity, this paper only considers parallel projection. However, the extension of our method to perspective projection is straightforward.

The starting point of the Shear-Warp algorithm is the viewing transformation matrix M_{view}, which transforms points from object space to image space. Lacroute decomposes this matrix into shearing and warping matrices:

$$M_{view} = M_{warp2D} \cdot M_{shear3D} \tag{1}$$



(a) Intermediate image (b) Final (warped) image

Fig. 2. In the Shear-Warp algorithm, the volume slices are first sheared and combined, yielding an distorted intermediate image (a). The final image (b) is obtained by warping the intermediate image.

The following subsections describe the derivation of those matrices from the view transformation matrix.

A. Shearing

The matrix $M_{shear3D}$ transforms the object space to the sheared object space, by shearing the coordinate system until the viewing direction becomes perpendicular to the slices of the volume. In image space, the viewing direction vector is always $\mathbf{v_i} = (0, 0, 1)$. Therefore, the actual view direction $\mathbf{v_o}$ is defined as:

$$\mathbf{v_i} = M_{view,3\times3} \cdot \mathbf{v_o} \tag{2}$$

where $M_{view,3\times3}$ is the upper-left 3×3 submatrix of M_{view}. Hence, $\mathbf{v_o}$ is defined as:

$$\mathbf{v_o} = \begin{bmatrix} m_{12}m_{23} - m_{22}m_{13} \\ m_{21}m_{13} - m_{11}m_{23} \\ m_{11}m_{22} - m_{21}m_{12} \end{bmatrix} \tag{3}$$

where the m_{ij} are the elements of M_{view}. In the following, we assume that $\mathbf{v_o}$ is mostly parallel to the $+z$ axis (i.e. $\mathbf{v_0}.z > \mathbf{v_0}.x \wedge \mathbf{v_0}.z > \mathbf{v_0}.y$). In the $x - z$ (resp. $y - z$) plane the slope s_x (resp. s_y) of $\mathbf{v_o}$ is the ratio of the x (resp. y) and z components of $\mathbf{v_o}$. Therefore, the shear necessary in the x (resp. y) direction is the negative of this slope:

$$s_x = -\frac{\mathbf{v_0}.x}{\mathbf{v_0}.z} \tag{4}$$

$$s_y = -\frac{\mathbf{v_0}.y}{\mathbf{v_0}.z} \tag{5}$$

The shearing matrix is then defined as:

$$M_{shear3D} = \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6}$$

The volume slices transformed by this matrix can be composited together along the $+z$ axis, yielding the distorted *intermediate image*. The final image is obtained by warping this intermediate image.

B. Warping

Lacroute [3] defines the warping matrix as:

$$M_{warp2D} = M_{view} \cdot M_{shear3D}^{-1} \tag{7}$$

$$= M_{view} \cdot \begin{pmatrix} 1 & 0 & -s_x & 0 \\ 0 & 1 & -s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

This algorithm is originally designed for interactive rendering of volume data on the CPU. However, this algorithm can benefit from the versatility and computational power of graphics hardware for improved performance.

III. CPU-based Shear-Warp

We have implemented the Shear-Warp method on the CPU. The volume data have been compressed using the RLE scheme as follows. For each coordinate axis (corresponding to a principal view direction) and for each slice, we compress each scan line of voxels once having classified the voxels using the Hounsfield [10] scale (quantitative scale for describing radiodensity) and a transfer function (linear curve with a lower and an upper bounds) wich converts the input data into opacities. As a result, for each scan line of voxels we get a list of runs. There are two types of run: completely transparent (considered as subsets of empty voxels) and not completely transparent. This process results in 3 data structures, one for each coordinate axis. Each data structure is a set of lists of runs, each list corresponds to one scan line of one slice. During volume rendering, these data structures are traversed slice by slice and line by line, and the completely transparent runs are ignored (empty voxels are skipped), which speeds up rendering drastically. The algorithm does not make use of any 3D data structure such as octree as suggested in the original paper on Shear-Warp [3].

IV. GPU-Based Shear-Warp

Our aim is to use the computational power of graphics processors to speed up the rendering process of the Shear-Warp algorithm. The power of the graphics processors comes from a highly parallel structure, along with built-in mathematical operations such as matrix product and texture filtering. Recently, the graphics processors have become more versatile by allowing the user to customize the operations performed at the vertex level (*vertex shader*) and at the fragment level (*fragment shader*). Those user-defined programs are then executed in parallel, yielding high quality results in realtime.

Our approach uses the programmability of graphics hardware to render volume data using the Shear-Warp algorithm. To this end, we first store the volume data into a 3D texture, and we generate a set of n quadrilaterals along the z axis, where n is the number of slices (Fig. 3). The texture coordinates of each vertex of each slice are defined as shown in Fig. 4. Using those coordinates, the n^{th} quadrilateral represents the n^{th} slice.

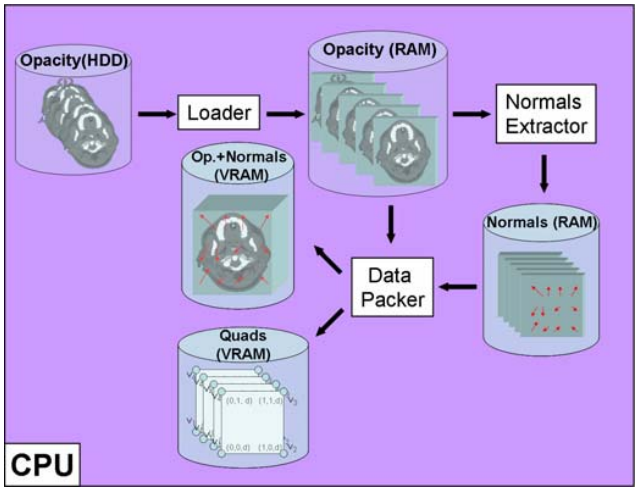


Fig. 3. In a preprocessing step, our algorithm loads the opacity images from the hard disk, and estimates the normals for each voxel. Finally, the opacity and normals are stored within the GPU memory as a 3D texture.

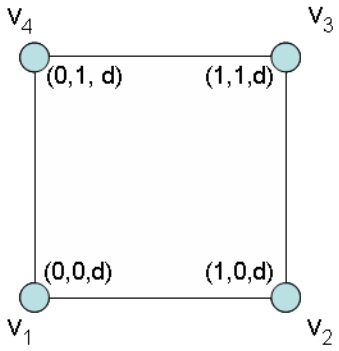


Fig. 4. Each slice is rendered as a textured quadrilateral, where the texture coordinates of each vertex contains a value $d = sliceNumber/sliceCount$, representing the slice index in the 3D texture.

In OpenGL, the view transformation matrix M_{view} is the product of the *modelview* and *projection* matrices:

$$M_{view} = M_{GLModelView} \cdot M_{GLProjection} \quad (9)$$

Using M_{view}, the shearing matrix $M_{shear3D}$ can be obtained as described in the previous section. This matrix is sent to the GPU Vertex Shader for shearing each slice according to the view direction. Then, the fragment shader fetches the opacity corresponding to each voxel of each slice from the 3D texture. The compositing of slices can then be obtained by drawing each slice from back to front using hardware alpha-blending (Fig. 5).

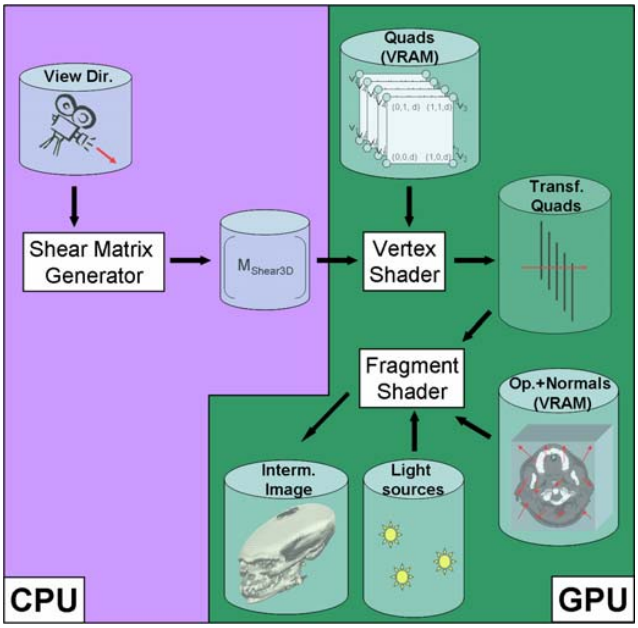


Fig. 5. The first rendering pass consists in applying the shear matrix to the volume slices (represented as a set of quadrilaterals) using the GPU vertex shader. Then, each fragment of those quadrilaterals is shaded on the GPU using the Phong model (Fig. 7), according to the corresponding opacity and normal. The output of the fragment shader is the intermediate, distorted image.

This step yields the distorted, *intermediate image*. The warping step is performed by applying the warping matrix M_{warp} to a textured quadrilateral representing the intermediate image (Fig. 6).

The volume data used in our work comes from medical imaging systems, providing an opacity value for each voxel. Even though this opacity could be used directly to generate images of the volume (Fig. 8(a)), the volumetric appearance can be enhanced by properly shading the volume according to a virtual light source. Our volume renderer uses the Phong [11] model to compute the outgoing radiance at a given point $\mathbf{p}$ with normal $\mathbf{n}$ (Fig. 7):

$$L_o(\mathbf{p}, \omega_o) = k_a + \frac{k_d}{\pi} L_i(\mathbf{p}, \omega_i) \omega_i \cdot \mathbf{n} + k_s (R(\omega_i, \mathbf{n}) \cdot \omega_o)^n \quad (10)$$

where:

- ω_i and ω_o are the incoming and outgoing directions
- k_a is the *ambient lighting*
- k_d and k_s are respectively the diffuse and specular terms of the reflectance function
- $R(\omega_i, \mathbf{n})$ is the direction of perfect reflection of ω_i with respect to $\mathbf{n}$.

However, the evaluation of this model requires the knowledge of the normal at each point. As a precomputation, we extract the normal at a voxel (x, y, z) from the neighboring opacity values $op(x \pm 1, y \pm 1, z \pm 1)$ using a simple gradient

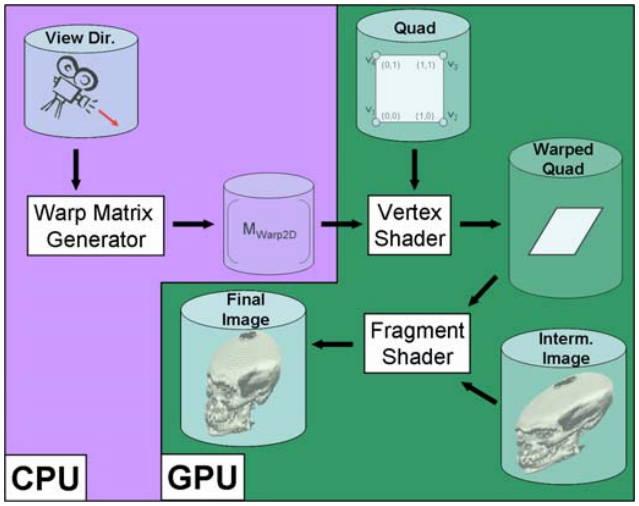
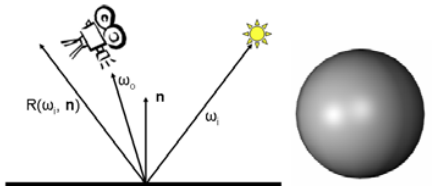


Fig. 6. In the second rendering pass, the intermediate image is warped to eliminate the distortions. This task is performed by applying the warping matrix to a single quadrilateral in the GPU vertex shader. Then, the GPU fragment shader maps the intermediate image onto the warped quadrilateral, yielding the final image.



(a) (b)

Fig. 7. (a) The notations used in the Phong lighting model. (b) A sphere rendered using this model.

approach:

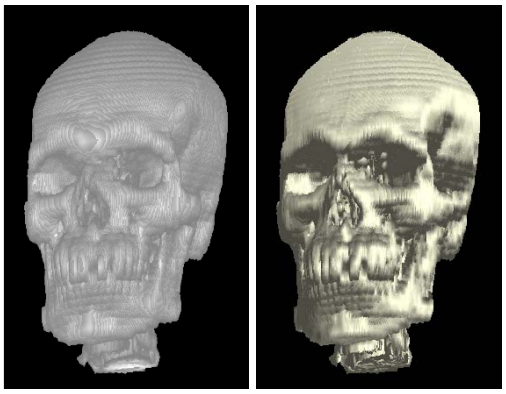
$$\mathbf{n}(x,y,z) = \frac{1}{2}\begin{pmatrix} op(x+1,y,z) - op(x-1,y,z) \\ op(x,y+1,z) - op(x,y-1,z) \\ op(x,y,z+1) - op(x,y,z-1) \end{pmatrix} \quad (11)$$

Using this simple normal estimation along with the Phong lighting model, the obtained results exhibit a more convincing volumetric appearance (Fig. 8(b)).

V. GPU-Based Ray Marching

Unlike the Shear-Warp method, the ray marching algorithm generates the final image in a single pass. Due to the versatility of recent graphics hardware, the costly ray marching can be performed very efficiently within the fragment shader, as described below.

In a preprocessing step, the volume data are read from the hard disk, and the normals are estimated as in the previous section. However, this method do not rely on volume slicing any more. Instead, the volume is represented by a simple cube (Fig. 10) and considered as a 3D texture.



(a) Opacity only (b) Phong shading

Fig. 8. Compared to the direct visualization of the opacity values (a), the Phong shading drastically improves the visual perception of volumes (b).

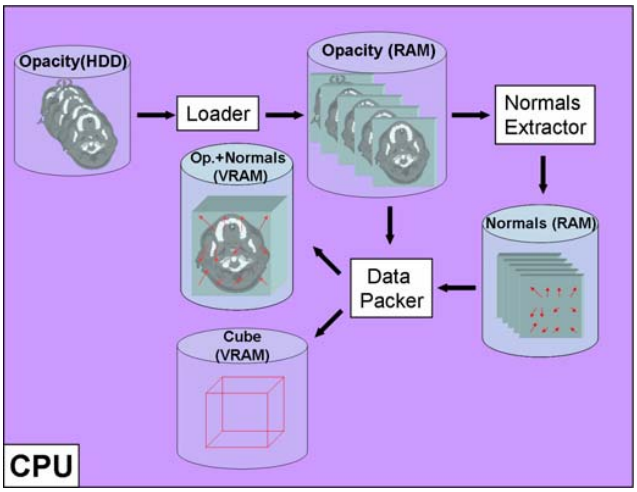


Fig. 9. As for the Shear-Warp algorithm the opacity values are read from the hard disk and the normals are estimated. This data is stored within 3D textures. However, for ray marching the volume is no longer represented by slices, but by a simple cube.

The ray marching algorithm then works as follows: for each pixel covered by the cube in the final image, the fragment shader traverses the 3D texture to determine the location of a point with a sufficient opacity (Fig. 10). To maintain performance, the fragment shader samples the volume data uniformly at a fixed sampling rate from front to back. If the opacity at a sample point is above a given threshold, the traversal is terminated and the lighting at this point is returned. However, as point sampling along the ray is uniform, aliasing artifacts may appear when the sampling rate is too low (Section VI-C).

The overall algorithm for GPU ray marching is illustrated in Fig. 11. The computational load is completely moved onto the GPU, and provides interactive frame rates even with several hundreds of samples. Furthermore, the reader should note that the ray marching does not require the definition of a

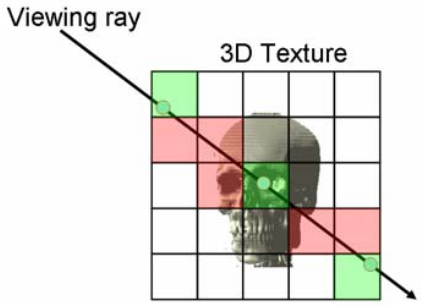


Fig. 10. Ray Marching algorithm on the GPU: for a given direction, the fragment shader uniformly samples the volume data to determine the closest point with sufficient opacity. The voxels of actual evaluation of the opacity are filled in green in this figure. The red voxels represent voxels traversed by the ray, but where the opacity is not evaluated due to an insufficient sampling rate. The resulting aliasing artifacts are clearly visible in Section VI-C.

particular projection axis. Therefore unlike with the Shear-Warp algorithm, the user can rotate smoothly around the object without seeing "popping" artifacts.

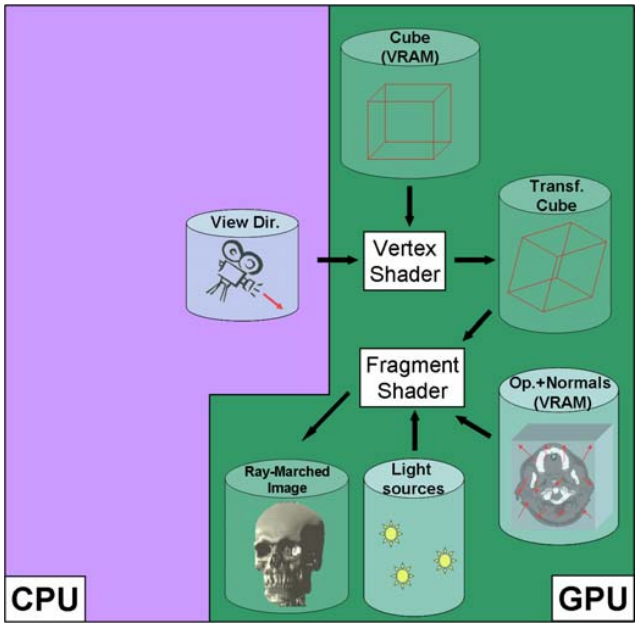


Fig. 11. The Ray Marching algorithm is entirely implemented on graphics hardware. Provided a camera position the vertex shader transforms the cube accordingly. For each pixel covered by the cube, the fragment shader traverses the volume data as shown in Fig. 10. The output of the fragment shader is the final image of the volume.

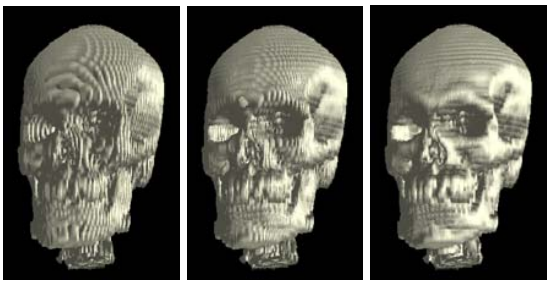
VI. Results

We have experimented with our 3 volume rendering methods (CPU-based and GPU-based Shear-Warp, GPU-based Ray Marching) on a 3.8Ghz Pentium 4 equipped with 2GB RAM and two nVidia GeForce FX7950 GX2 512Mb running in *split frame* SLI mode.

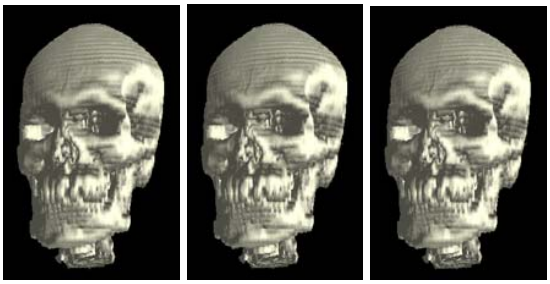
The input images are of scanner type (TDM) and saved in files in DICOM format. The input data files are read using the DCMTK library. Details on DICOM format and DCMTK can be found in [12]. The conversion of the input data into opacities is performed using the Hounsfield scale [10] and a transfer function which is linear and bounded by two lower and upper values.

A. GPU-Based Shear-Warp

1) Skull: ($320 \times 320 \times 46$ voxels). Our GPU-based renderer renders this volume (made up of 46 slices) at 95 fps. However, the slices are clearly visible (Fig. 13(a)). To overcome this problem, we increase the sampling rate of the volume by virtually inserting slices by interpolation (Fig. 12). In other words, we compute a certain number of additional slices between two successive input slices using linear interpolation. This number is called, from now on, sampling rate. Even though this decreases the frame rate (Table I), the volumetric appearance of the model is clearly improved. Furthermore, the model can still be rendered interactively even with 20 times more slices, which corresponds to the rendering of a virtual model composed of 920 slices.



(a) $s = 1$ (b) $s = 2$ (c) $s = 5$



(d) $s = 10$ (e) $s = 15$ (f) $s = 20$

Fig. 12. Rendering of the *Skull* model, where s is the sampling rate. The image quality is significantly improved by applying a sampling rate of 5. However, the quality improvement is hardly noticeable using higher sampling rates.

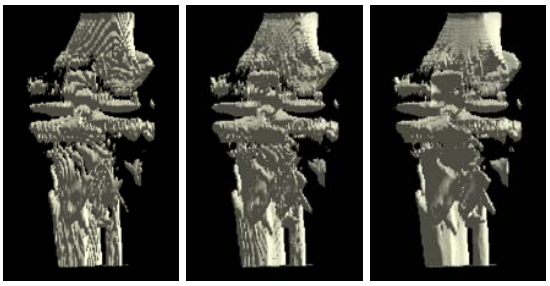
2) Knee: ($512 \times 512 \times 73$) This volume is rendered by our system at 134 fps with Phong shading (Fig. 13). As in the previous example, the frame rate drops linearly with the number of additional slices.

TABLE I
FRAME RATES PER SECOND OBTAINED WITH THE GPU-BASED METHOD FOR THE SKULL MODEL (46 SLICES) AND USING SEVERAL SAMPLING RATES.

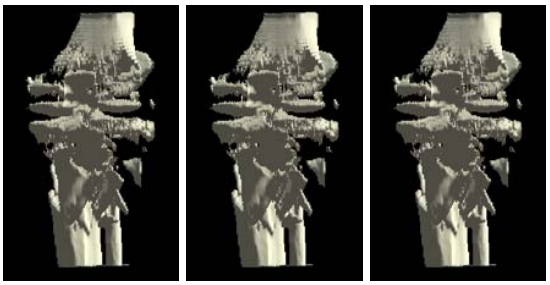
Sampling rate	Opacity only	Phong shading
1	199 fps	95 fps
2	104 fps	49 fps
5	43 fps	20 fps
10	22 fps	10 fps
15	15 fps	7 fps
20	11 fps	5 fps

TABLE II
RENDERING TIMES IN SECONDS FOR THE SKULL AND KNEE MODELS OBTAINED WITH THE CPU-BASED METHOD AND USING UP TO 5 SAMPLING RATE VALUES.

Sampling rate	Skull	Knee
1	22	110
2	43	155
3	64	226
4	86	300
5	107	370



(a) $s = 1$, 134 fps (b) $s = 2$, 70 fps (c) $s = 5$, 29 fps



(d) $s = 10$, 15 fps (e) $s = 15$, 10 fps (f) $s = 20$, 7 fps

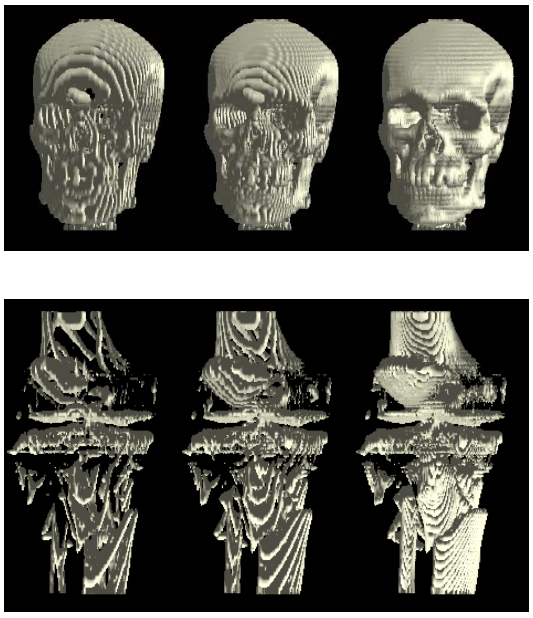
Fig. 13. Rendering of the *Knee* model, where s is the sampling rate.

B. CPU-Based Shear-Warp

We used the same models as in VI-A. Phong model has been used for direct lighting. The results obtained with the CPU-based Shear-Warp method are given in table II. This table shows volume rendering times rather than frames per second because the method is from far slower than the GPU-based method. This is why we used only a maximum sampling rate equal to 5. It is clear that the GPU-based method outperforms the CPU-based one.

C. GPU-Based Ray Marching

The same models were rendered using the GPU Ray Marching algorithm described in the previous Section. Even though the models have different complexities ($320 \times 320 \times 46$ =4.5MVoxels for the *Skull* model, and $512 \times 512 \times 73$ =18.25MVoxels), the rendering times only depend on the number of marching steps (that is the number of sample points along a viewing ray) in the volume (Table III). Furthermore, the high performances are due to the efficient support of dynamic branching in the fragment shader: for a given ray, its traversal is terminated as soon as the target opacity is reached. Due to the high spatial coherency of the 3D volumes (two neighboring voxels are likely to have a similar opacity), the rays corresponding to neighboring pixels in the image are likely to terminate after a similar number of steps. Therefore, the dynamic branching in the fragment shader can be used efficiently without harming the parallel performance of graphics hardware.



(a) 50 steps (b) 100 steps (c) 300 steps

Fig. 14. The *Skull* and Knee models rendered using Ray Marching on graphics hardware with several numbers of marching steps.

VII. CONCLUSION

In this paper, we presented a method for efficient volume rendering using the shear-warp algorithm on the GPU. To our knowledge, this is the first implementation on the GPU. The volume data at our disposal contain no more than 46 slices for the *Skull* volume data set and 73 for the *Knee*

TABLE III
FRAME RATES OBTAINED BY GPU RAY MARCHING OF THE *Skull* AND *Knee* MODELS.

Marching steps	*Skull*	*Knee*
50	308 fps	311 fps
100	175 fps	175 fps
200	95 fps	94 fps
300	64 fps	64 fps
400	48 fps	48 fps
500	44 fps	44 fps

model, say few slices. As shown in Fig. 12(a), these slice numbers are not sufficient for getting an acceptable quality of the rendered images. To overcome this problem, we have proposed to add in-between slices computed by interpolation. This has provided better results as shown in Fig. 12(b-f). When the sampling rate is equal to 10, the total number of slices is 420 and the frame rate is 10 fps for the *Skull* model illuminated using the Phong model (Table I). It has clearly been shown that the GPU-based Shear-Warp is far faster than the CPU-based Shear-Warp.

We have also implemented on the GPU the ray marching algorithm. To cope with the problem of insufficient volume slices, one can increase the number of marching steps (sample points along a viewing ray) as shown in Fig. 14. If this number is equal to 400, the obtained frame rate is 48 fps for the *Skull* model illuminated using the Phong model (Table III). A number of marching steps equalling 400 corresponds to a total number of 400 slices (input and interpolated slices) for a shear-warp algorithm. Consequently, it is clear that GPU-based ray-marching (48 fps) is faster than the shear-warp algorithm (10 fps).

We think that the GPU-based ray-marching algorithm outperforms the shear-warp algorithm in terms of speed-up and image quality. Indeed, as it is based on ray tracing, more sophisticated reflection and scattering models could be straightforwardly used to increase the quality and the interpretation of the rendered images of the volume data.

REFERENCES

[1] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, 1988.
[2] K. Subramanian and D. Fussel, "Applying space sudivision techniques to volume rendering." IEEE Symposium on Data Visualization, 1990, pp. 150–159.
[3] P. Lacroute, "Fast volume rendering using a shear-warp factorization of the viewing transformation," Ph.D. dissertation, Stanford University, 1995.
[4] K. Engel, M. Kraus, and T. Ertl, "High quality pre-integrated volume rendering using hardware-accelerated pixel shading."
[5] C. Rezk-Salam, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization."
[6] *Real-Time Volume Graphics*. A.K. Peters, 2006.
[7] H. KELES, A. ES, and V. Isler, "Acceleration of volume rendering with programmable graphics hardware," *The Visual Computer*, 2007.
[8] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering."
[9] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Stasser, "Smart hardware-accelerated volume rendering."
[10] S. Hounsfield, "e2phy.in2p3.fr/2002/actes/lisbona.doc."
[11] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, pp. 311–317, 1975.
[12] DCMTK, "http://dicom.offis.de/dcmtk.php.en."