GPGPU Based Non-photorealistic Rendering of Volume Data

Anca Morar, Florica Moldoveanu, Victor Asavei, Lucian Petrescu, Alin Moldoveanu, Alexandru Egner

The Faculty of Automatic Control and Computers, University "POLITEHNICA" of Bucharest,Romania,
(e-mail:{anca.morar, florica.moldoveanu, victor.asavei,
lucian.petrescu, alin.moldoveanu, alexandru.egner}@cs.pub.ro)

Abstract: Nowadays, non-photorealistic volume rendering has become a useful technique in medicine and scientific visualization. One of these rendering techniques is silhouette extraction of iso-surfaces. This paper proposes three methods of extracting silhouettes from relatively large datasets very fast (in some cases, even in real time), using the GPGPU technology. These methods are suitable for different types of datasets, applications and hardware characteristics. The first method extracts the iso-surface and then computes its silhouette. The second one extracts only the silhouette and computes the visibility of each contour vertex using an algorithm inspired by ray casting. The third method uses a CUDA rasterizer in order to render iso-surfaces and silhouettes from large datasets.

Keywords: Computer vision, Parallel algorithms, Non-photorealistic volume rendering, Marching cubes, Geometry shader

1. INTRODUCTION

Volumetric visualization has become a widely used technique in many domains, especially in medicine, for diagnosis. The datasets acquired from medical devices (CT or MRI devices), in the form of stacks of slices representing sections through the scanned body, are processed in order to obtain different interpretations of the scanned objects.

In addition to the usual volume rendering techniques (e.g. direct volume rendering, MIP, indirect volume rendering), a series of new illustrative visualization methods have been designed in order to highlight different important characteristics in a volume dataset.

The non-photorealistic rendering methods presented in this paper are based on the marching cubes algorithm by (Lorensen and Clinne 1987), and the ideas described in (Burns et al. 2005). The applications render the silhouette of an iso-surface extracted from a volume dataset, depending on the viewer's position, as shown in Figure 1. The visualization of a body part's silhouette is very important in medicine, because it concentrates the attention on the salient elements of the scanned object, and not on the texture, colour, or lighting of the scene that can, in some cases, complicate too much the understanding of the object's medical characteristics. Although medicine is the most significant domain where volume rendering (and in this case, non-photorealistic volume rendering) is useful, there are other fields, like education (biology or medical training) and scientific visualization, where this technology leads to very interesting results.

The next section presents the state of the art in this domain, in order to understand the challenges of volume rendering. The third chapter describes the marching cubes algorithm with CUDA and the silhouette extraction technique by (Burns et al. 2005) that inspired the research presented in this article. The fourth chapter describes the algorithms behind the silhouette rendering methods. The fifth section is dedicated to presenting some results regarding computing times and visual effects. The final section points out several conclusions.

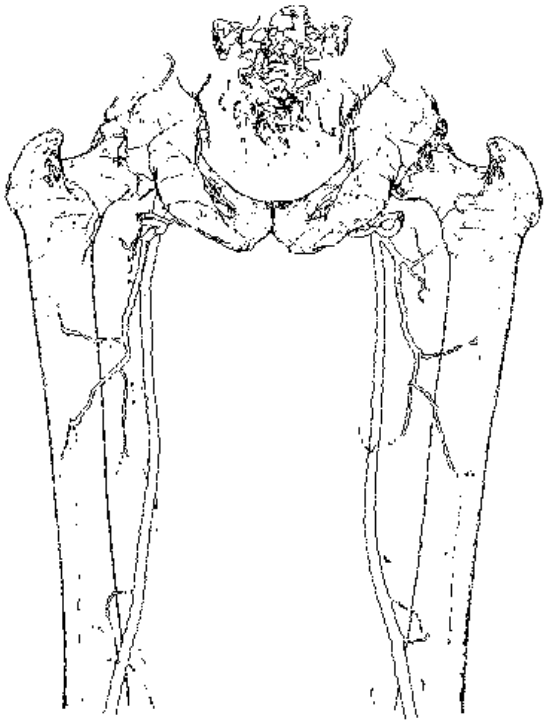

Fig. 1. Non-photorealistic rendering of femoral and pelvic bones from a CT dataset of size 512^3 using marching cubes and geometry shader. The dataset was obtained from Floreasca Emergency Hospital, Bucharest.

All the tests presented in the following chapters were made on a GT 230M GPU with 1GB GPU RAM.

2. STATE OF THE ART

One of the most popular techniques for volume rendering, marching cubes, by (Lorensen and Cline 1987), approximates an iso-surface through a triangular mesh, by computing its intersection with the volume cells or voxels. This method obtains the geometry of the iso-surface, unlike volume ray casting described by (Levoy 1990) that casts rays from the eye to each image pixel into the volume, sampling it at certain intervals, and compositing the colours and opacities for every sample.

Some non-photorealistic volume rendering techniques use iso-surface reconstruction. (Pelt et al. 2008) propose an interactive GPU illustrative framework based on a particle system that is created by using a marching cubes like approach. (Dong et al. 2003) introduce volumetric hatching, a technique that produces pen-and-ink style images from medical volume data by generating iso-surfaces from the data with marching cubes and applying standard surface-hatching techniques. (Yuan and Chen 2004) illustrate surfaces by drawing feature lines, such as silhouettes, valleys, ridges and surface hatching strokes. This approach first extracts a point-based surface model described by (Co et al. 2003) and then renders the volume, embedding surfaces from the first step.

Other methods, based on the ray casting algorithm, modify the transfer function in order to obtain non-photorealistic renderings. (Luo et al. 2009) present an illustrative technique for focusing on a user-driven region of interest with a distance function which controls the opacity of the voxels. (Kindlmann et al. 003) advance the use of curvature information in multi-dimensional transfer functions, bringing contributions to three different application areas: non-photorealistic volume rendering, surface smoothing via anisotropic diffusion and visualization of iso-surface uncertainty. (Bruckner and Gröller 2007) propose the concept of style transfer functions that enables flexible data-driven illumination which goes beyond using the transfer function only for assigning colours and opacities. (Hadwiger et al. 2005) describe a real-time rendering pipeline for implicit surfaces defined by a volumetric grid using a ray-casting approach and local shape descriptors.

(Burs et al. 2005) render silhouettes from volume data with a voxel-based marching lines technique that traces contours. This method will be described in more detail in the next chapter.

3. RELATED WORK

This section focuses on the previous techniques that represent the basis for this new GPGPU silhouette rendering methods.

3.1. Silhouette extraction from volume data

In their paper, (Burns et al. 2005) propose a CPU method for extracting silhouettes and suggestive contours. They extract the silhouette by intersecting the iso-surface with the points where the normal is perpendicular to the view ($\overline{N} \cdot \overline{V} = 0$). Figure 2 presents an example of the silhouette extraction algorithm – for a volume voxel.

Based on the scalar values of the voxel's vertices (A_0, A_1 and A_3 are outside the iso-surface because their scalar values a_0, a_2, a_3 < *iso-value*, and A_1 is inside the iso-surface because a_1 > *iso-value*), a triangle is generated, with vertices B_0, B_1 and B_2 and their normal vectors $\overline{N}_0$, $\overline{N}_1$ and $\overline{N}_2$.

Let $\overline{V} = eye - B$ be the view vector for vertex B. The dot product $d = \overline{N} \cdot \overline{V}$ is then computed for every triangle vertex. If the dot products have different signs (for example, d_0 and d_1 are positive, d_2 is negative), this means that there is a silhouette segment ($[C_0C_1]$) inside the triangle. The silhouette segment is determined using linear interpolation from the triangle vertices, based on d_0, d_1 and d_2.

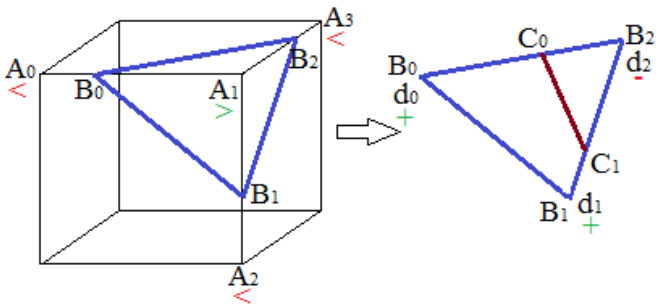

Fig. 2. Extraction of a silhouette segment from an iso-surface, based on its intersection with a volume voxel.

After obtaining the lines from the volume, a visibility test is realized by casting rays from each silhouette segment endpoint to the eye. For each volume voxel intersected by a ray, they determine whether it intersects the iso-surface in that cell by examining the intersection with the face by which it leaves the cell. At that intersection, they use linear interpolation from the four corners to determine the function value and see if it has changed sign (indicating that the ray has passed from outside to inside the iso-surface). If it has, the originating vector is marked as occluded.

3.2. GPGPU parallelism

This sub-section describes several characteristics of the GPGPU paradigm, as presented in the CUDA Programming Guide from Nvidia (2011a). GPGPU refers to using the GPU for other computations besides graphics rendering.

CUDA is a parallel computing architecture that leverages the parallel compute engine in Nvidia GPUs to solve complex computational problems in a more efficient way than on the CPU. At the core of CUDA there are three main abstractions that are exposed to the programmers as a set of language extensions: a hierarchy of threads, shared memory and barrier synchronization.

CUDA C extends the well-known C programming language by allowing the definition of kernels that are executed in parallel by different threads, unlike the regular C functions that are executed only once. The threads are organized into blocks and the blocks into grids, allowing programmers to organize the problems depending on the size of the data to be handled.

CUDA threads can access data from different memory spaces during their execution. The GPU memory can be local, shared, global, texture or constant.

Threads within a block can cooperate by sharing data through shared memory and by synchronizing the execution in order to coordinate the access to memory. Through barrier synchronization, threads in a block wait for all the other ones to finish before proceeding with the next computation.

One research focus of our group is to develop parallel algorithms for analysis and visualization in medicine based on the GPGPU paradigm. As a result, several improvements to CUDA implementations of marching cubes, Canny edge detector and Hough transform were proposed by (Petrescu et al. 2011; Morar et al. 2012). Other algorithms that use the power of the GPUs for parallel computation are described in the current paper.

3.2. Marching cubes with CUDA

The silhouette rendering algorithms presented in this paper follow a parallel implementation, using CUDA kernels. A CUDA kernel is defined for one data element and then executed in parallel for all the data elements, for example, for all the voxels in a volume dataset. The first silhouette rendering technique is very fast, but uses a lot of GPU memory. This is why other two algorithms that would require less video memory have been designed. All three methods are based to some extent on the CUDA implementation of marching cubes.

The parallel implementation of marching cubes from the (Nvidia 2011a, b) leads to very satisfactory results, compared to the CPU implementation. For a dataset of size 256^3, the application runs at 10 fps if the iso-surface is reconstructed at every frame, unlike the CPU application that runs at 0.03 fps. If the reconstruction of the surface is made only when the iso-value has been changed, the application runs at 60 fps (while the CPU application runs at 0.2 fps).

The marching cubes algorithm (in pseudo code) from the CUDA SDK example is presented below. There is an alteration to the initial algorithm. In the classic implementation, the normal of the triangles that approximate the iso-surface are computed based on the triangle vertices using the cross product. Thus, only one normal is generated for each triangle. In the proposed method, the triangle normals are replaced by gradient vectors. The gradient vectors of the voxels' vertices are approximated with finite differences based on the scalar values within the volume. The gradient vectors of the iso-surface triangles are computed using linear interpolation from the gradient vectors of the voxel's vertices.

This alteration offers a better visual effect of the rendered iso-surface by allowing smooth illumination and is also important for the further extraction of silhouettes. Since the silhouette extraction presented in section 3.1 is based on the difference between the normal vectors of a triangle's vertices, the original algorithm, with a single normal for the entire triangle cannot be used.

Pseudo code 1
Marching cubes with CUDA

```
Step 1.  copy data to the GPU
   copy the search tables and the volume vol
into texture memory on the GPU.
Step 2. classify and remove non-border voxels
   for every voxel v in vol do (in parallel
using a CUDA kernel)
      classify v (interior/exterior/border)
   end for
   read total number of active (border) voxels
   for every voxel v in vol do (in parallel
using a CUDA kernel)
      if v is border voxel then
         save v in a vector of active voxels
      end if
   end for
Step3. compute triangles of the iso-surface
   for every active voxel av do (in parallel
using a CUDA kernel)
      for every vertex vx of av do
         read value of vx (from texture)
         read value of neighbouring vertices
(for the approximation of the gradient vector)
         compute gradient at vx (with finite
differences)
      end for
      for every intersection point ip of the
iso-surface with the edges of v do
         determine value of ip and normal at ip
(using linear interpolation)
      end for
      for every triangle t of the iso-surface do
         save vertices and normals into vertex
buffer objects (VBOs - on the GPU, in order to
be directly rendered by the GPU)
      end for
   end for
End of pseudo code
```

Changing the viewer's position does not affect the performance of this rendering method. This is why the first approach presented in this paper is based mainly on the implementation of marching cubes with CUDA.

Figure 3 presents the surface of a femoral bone and a part of the pelvis extracted with marching cubes from a volume dataset of size 256^3.

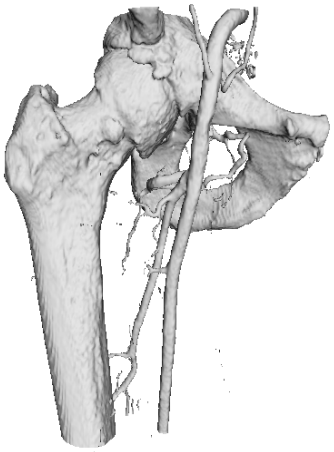

Fig. 3. Surface rendered using the CUDA marching cubes algorithm. The dataset was obtained from Floreasca Emergency Hospital, Bucharest.

In order to handle large datasets, the original volume can be divided into smaller sub-volumes (chunks) that are reconstructed serially on the GPU using the same marching

cubes CUDA kernel, as described by (Petrescu et al. 2011). Instead of having two VBOs, one for the triangle vertices and one for their normal vectors, there will be two VBOs for each sub-volume. This implementation is slightly slower than the original marching cubes with CUDA, but can handle considerably larger datasets.

4. SILHOUETTE RENDERING USING CUDA

4.1. Silhouette rendering with marching cubes and geometry shader

This new approach takes advantage of the programmable rendering pipeline's flexibility. After extracting the triangular mesh approximating the iso-surface, the silhouettes are rendered using a vertex shader and a geometry shader.

The input of the vertex shader contains the position and normal for each vertex, the eye position in world coordinates, the model matrix and the model-view-projection matrix. The output of the vertex shader consists of the vertex positions in clip space, the position and normal vector and the eye position in world coordinates.

While the vertex shader can process only vertices, the geometry shader can handle whole primitives. The input of the geometry shader is represented by the mesh triangles in world and clip coordinates and the eye position in world coordinates. The output contains either nothing, or a segment of the silhouette iso-surface in clip coordinates, computed based on the silhouette extraction algorithm depicted in Figure 2. For every triangle of the iso-surface which intersects the surface where the normal is perpendicular to the view, the geometry shader emits a silhouette segment representing that intersection.

In order to compute the visibility of the silhouette points, the geometry representing the iso-surface is sent to the graphics pipeline, and rasterized only for the creation of the depth buffer. After this step, the contour lines are also rasterized by updating both the colour and the depth buffer. In consequence, only the silhouette pixels that are not occluded are rendered.

Figure 4 shows the output of the geometry shader that represents the silhouette rendered without any visibility test, and the output of the presented algorithm, i.e. the silhouette rendered with the visibility test.

Fig. 4. Silhouette rendered without visibility test (left) and with visibility test (right).

The silhouette rendering flow described in this section is illustrated in Figure 5.

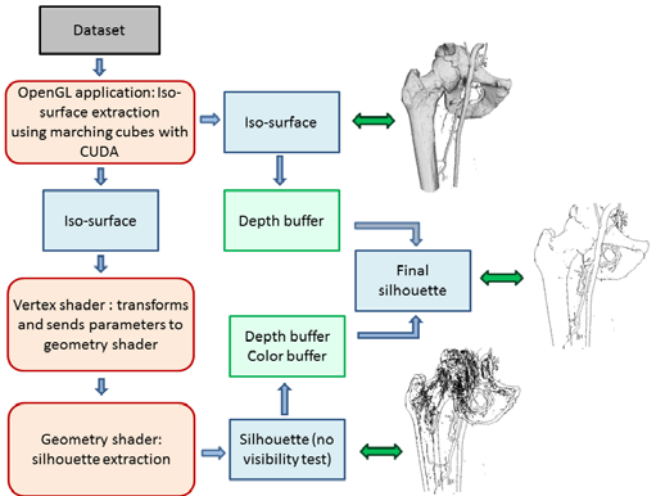

Fig. 5. The flow of the algorithm that renders the silhouette with marching cubes and geometry shader.

This method is the fastest, but requires a lot of GPU memory space. Storing the volume and the geometry on the GPU implies the use of small datasets for this implementation (up to 256^3). Still, this method can be used for larger datasets, but with an improvement.

In order to handle large datasets, an approach similar to the one proposed by (Petrescu et al. 2011) can be considered. The geometry represented by two VBOs for each sub-volume is further processed by the vertex and geometry shader in the same way as the method for smaller datasets. This approach solves to some extent the memory limitation, because, at one moment in time, only a small part of the volume is computed. But the need to store the whole geometry on the graphics card still limits the size of the dataset.

4.2. Silhouette rendering with ray based visibility test

The second approach, implemented on the GPU, proposes a visibility test based on the CPU silhouette rendering method of (Burns et al. 2005). First, it extracts the silhouette without any visibility test, and then, for every vertex belonging to the silhouette, it casts a ray through the volume to the eye position, intersecting it with the iso-surface and deciding whether the silhouette point is occluded or not. The steps of the algorithm are described in Figure 6.

The algorithm for silhouette rendering with ray based visibility test is described in the following pseudo code:

```
Pseudo code 2
Silhouette rendering with ray based visibility test
Step 1. similar to Step 1 in pseudo code 1
Step 2. similar to Step 2 in pseudo code 1
Step 3. extract silhouette lines that are visible
   for every active voxel av from volume vol do (in parallel using a CUDA kernel)
      determine the triangles approximating the iso-surface (similar to Step 3 in pseudo code 1)
      for every triangle t of the iso-surface do
         for every vertex vx do
```

```
            compute the view vector vv
            compute the dot product dp between
normal at vx and vv
         end for
         if dp for one vertex has a different
sign from dp for the other two vertices then
            compute the intersection of t with
the surface where the normal is perpendicular to
the view (based on the algorithm from section
2); the result is a silhouette segment s
            for each endpoint p of s do
               p is considered visible
               cast a ray r from eye to p
               determine where r enters vol
               for every ray step (the step size
ia equal to the size of a voxel edge) do
                  determine the next intersected
voxel iv (the voxel that contains the next point
on the ray) by incremental computing
                  compute the intersection of iv
with r and determine the face f of iv by which r
leaves iv
                  if the value of f (computed
using linear interpolation from the corners of
f) has changed sign, then
                     p is occluded
                  end if
               end for
            end for
            if both endpoints of s are visible
then
               store s into a VBO
            end if
         end if
      end for
   end for
End pseudo code
```

The algorithm should work for larger datasets because only the silhouette is stored on the GPU (unlike the first approach that stores the iso-surface as well). However, the performance is altered by the changing of the viewer's position (each time the eye position is changed, the silhouette has to be computed all over again) and by the visibility test.

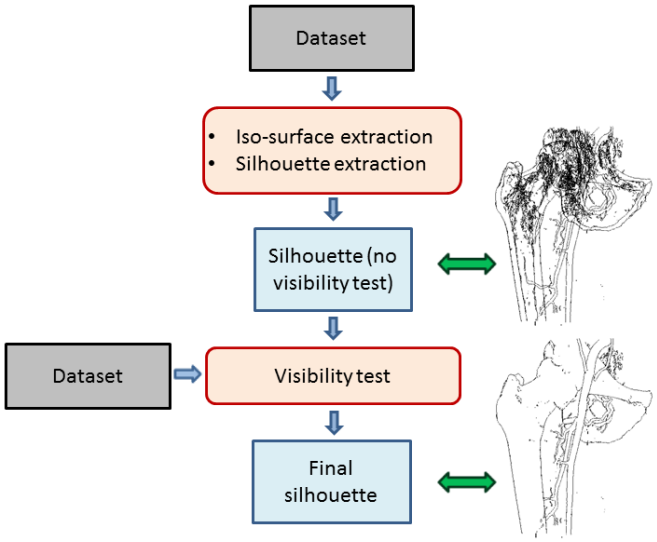

Fig. 6. The flow of the silhouette rendering algorithm with ray based visibility test.

Also, for larger datasets, the implementation using chunks is very useful. But this approach does not allow the generation of visible silhouette lines using only one CUDA kernel. Thus, `Step 3` from the previous pseudo code is divided into two steps. The first step, the generation of silhouette segments, both visible and occluded, is defined in a CUDA kernel. The kernel is executed for all the sub-volumes. The generated silhouettes are referred to as sub-silhouettes. The process is serialized, meaning that at one moment in time only one sub-volume is processed in parallel on the GPU. The second step, i.e., the visibility test, is defined in another CUDA kernel. This kernel is executed for all the sub-silhouettes, for every silhouette segment. The difference between the previous approach is that this time the visibility test of a sub-silhouette is run not only for the corresponding sub-volume, but for all the sub-volumes, serially.

The processing flow for handling large datasets is explained in Figure 7, in order to provide some insight into the silhouette rendering with ray based visibility test for large datasets.

4.3. Silhouette rendering using a CUDA rasterizer

A new approach that solves the problem of the memory limitation is proposed. It does not save the geometry, but renders it directly using a CUDA rasterizer. If there is GPU memory only to store the volume, an own rasterizer can be implemented, based on the z-buffer algorithm, using the CUDA architecture. Three buffers are used, one colour buffer, `cbuffer`, and two depth buffers (all three buffers have the size of the display window). The first depth buffer, `zbuffer1`, stores the maximum depth values for the triangles approximating the iso-surface. The second depth buffer, `zbuffer2`, stores the maximum depth values for the silhouette lines.

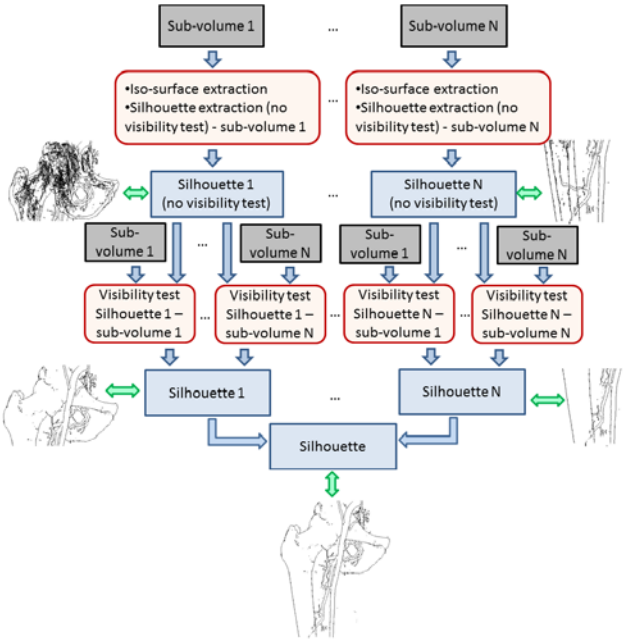

Fig. 7. The flow of the algorithm that renders silhouettes using a ray based visibility test – the approach that handles large datasets.

The pseudo code of the silhouette renderer with a CUDA rasterizer is presented below.

```
Pseudo code 3
Silhouette rendering with CUDA rasterizer
Step 1. copy volume to GPU
   save volume vol on GPU texture memory
Step 2. update depth buffers
   for every voxel v in vol do (in parallel
using a CUDA kernel)
      determine the triangles approximating the
intersection of v with the iso-surface
      for every triangle t (computed in
normalized device coordinates) do
         for every point (x,y,z) inside t or
belonging to its edges do
            if zbuffer1[x,y]<z then
               zbuffer1=z
            end if
         end for
         if the triangle contains a silhouette
line s then
            for every point (x,y,z) of s do
               if zbuffer2[x,y]<z then
                  zbuffer2[x,y]=z
               end if
            end for
         end if
      end for
   end for
Step 3. update colour buffer
   for every display window pixel (x,y) do (in
parallel using a CUDA kernel)
      cbuffer[x,y]=0
      if            zbuffer2[x,y]>=zbuffer1[x,y]
(silhouette pixel visible) then
         cbuffer[x,y]=1
      end if
   end for
End of pseudo code
```

The flow of the algorithm is described in Figure 8.

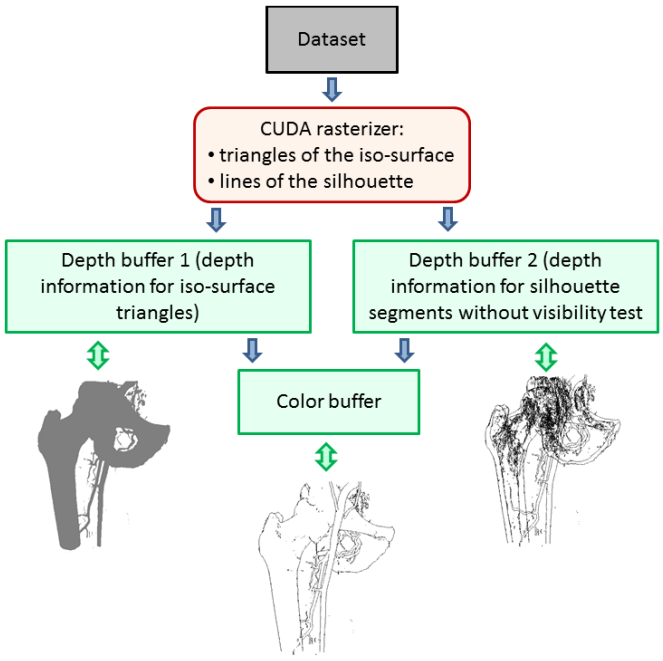

Fig. 8. The flow of the algorithm that renders silhouettes from volumes using a CUDA rasterizer.

The operation of updating the depth buffers is implemented with the CUDA `atomicMax()` operation in order to avoid read-modify-write hazards. The colour buffer is stored into a pixel buffer object (PBO).

This implementation is much slower than the first approach, because the CUDA rasterizer is not specialized for rendering, like the hardware rendering mechanisms of the GPU. Also, the generation of the visible silhouette segments has to be repeated every time the eye position is changed.

This CUDA rasterizer can be used also for indirect volume rendering. If there is not enough video memory to store the iso-surface extracted using marching cubes, the geometry can be rasterized directly with this new approach.

In order to handle larger datasets, the same depth buffer updating kernel can be run (`Step 2`), but for each sub-volume serially. The colour buffer updating kernel (`Step 3`) will be run only once, just like in the previous pseudo code.

5. RESULTS

Table 1 presents the computing times for rendering the silhouettes of iso-surfaces reconstructed from different volume datasets. The second row (with the header "M" from "mode") refers to the reconstruction mode: "a" states for the case when the reconstruction of the iso-surface/silhouette is made only when the iso-value is changed (does not depend on the changing of the viewer's position – is only applicable for the first rendering method); "b" states for the case when the reconstruction has to be done all the time (when the iso-value is changed or the eye position is changed). For each method, there are two columns, for the algorithm that handles small datasets and the one that can handle larger datasets.

Table 1. Computing times (ms) for the three methods of silhouette rendering (the case designed for small ("S") datasets and the one designed for larger ("L") datasets, with the division of the volume into sub-volumes).

Data vol.	M	1st alg S	1st alg L	2nd alg S	2nd alg L	3rd alg S	3rd alg L
32^3	a	4.2	5.75	x	x	x	x
32^3	b	7.4	8.79	10.3	12.76	14.5	15.5
256^3	a	16	29.8	x	x	x	x
256^3	b	115	117	677	839.7	501	515
512^3	a	x	66.1	x	x	x	x
512^3	b	x	865	x	4030	x	2609
512^2x1024	a	x	x	x	x	x	x
512^2x1024	b	x	x	x	x	x	9872

As can be observed, the first method is the fastest (especially when the reconstruction is made only when the iso-value is changed), but can handle only small datasets, because the whole geometry (iso-surface and silhouette) is stored on the GPU.

The second approach should work for larger datasets, but is limited by the memory allocations for the removal of non-border voxels and by the fact that the silhouette is stored on the GPU.

The third approach works faster than the second (for datasets of size 2563 and 5123), and can handle large datasets (theoretically, it can handle unlimited size volume datasets), but at non-interactive frame rates.

Each implementation uses a different visibility test, with more or less influence on the computing time. In order to observe this influence, tests have been made for each method, with or without the visibility test.

Table 2 presents the computing times for the silhouette rendering technique with marching cubes and geometry shader. The second column has the same meaning as the one in Table 1. The third and fourth columns correspond to the method (without and with visibility test) that does not divide the volume, being able to handle only small datasets. The last two columns correspond to the method that can handle larger datasets.

Table 2. Computing times (ms) for the first silhouette rendering method, using CUDA marching cubes and geometry shader, with and without visibility test.

Data vol.	M	Small no visibility	Small	Large no visibility	Large
32^3	a	4.2	4.2	5.7	5.75
32^3	b	7.4	7.4	8.6	8.79
256^3	a	15	16	29.2	29.8
256^3	b	114.5	115	175	177
512^3	a	x	x	63	66.1
512^3	b	x	x	815	865

The difference between applying the visibility test or not in the first silhouette rendering method is almost insignificant, because the iso-surface is extracted in any case. The small increase in speed for the approach that does not test the visibility of the silhouette vertices is caused by the fact that the iso-surface is sent only once to the graphics pipeline (to be further processed by the vertex and geometry shader in order to extract the silhouette), unlike the other approach, that sends the iso-surface twice, once for updating the depth buffer and once for silhouette extraction.

Table 3 presents the computing times for the silhouette rendering algorithm with ray based visibility test.

Table 3. Computing times (ms) for the second silhouette rendering method, with ray based visibility test.

Data vol.	small no visibility	Small	large no visibility	Large
32^3	6.7	10.3	10	12.76
256^3	121.5	677	143.4	839.7
512^3	x	x	782	4030

Here the differences between the two approaches, one with and one without visibility test, are remarkable, especially for the method that handles large datasets by dividing the volume into sub-volumes. Thus it can be concluded that the ray based visibility test is very time consuming, especially for large datasets.

Table 4 presents the computing times for the silhouette rendering algorithm that uses a CUDA rasterizer. The differences between applying the visibility test or not are also salient, but small compared to the differences in the second silhouette rendering approach. These differences are bigger than the ones in the first approach because, here, the iso-surface updates the depth buffer (or not, in case of no visibility test) using an own rasterizer, and not the specialized hardware rendering mechanisms of the GPU.

Table 4. Computing times (ms) for the silhouette rendering method with CUDA rasterizer.

Data vol.	Small no visibility	Small	Large no visibility	Large
32^3	11.2	14.5	15	15.5
256^3	490	501	334	515
512^3	x	x	2213	2609
512^2x1024	X	x	5143	9872

Also, the visibility test for the three approaches leads to different visual results: the visibility test for the first and third approach is made for every point belonging to a silhouette segment, while in the second approach, it is made only for the endpoints of a silhouette segment. This is why the silhouette rendered with the second method has more discontinuities (see Figure 9).

The three silhouette rendering methods have been compared not only regarding the computing times and the size of the datasets that can be handled, but also regarding the visual quality. Silhouettes from several datasets have been rendered with the three approaches and compared with manually drawn silhouettes. The tested datasets are shown in Figure 10. Table 5 presents the percentage of contour pixels correctly rendered for the proposed methods.

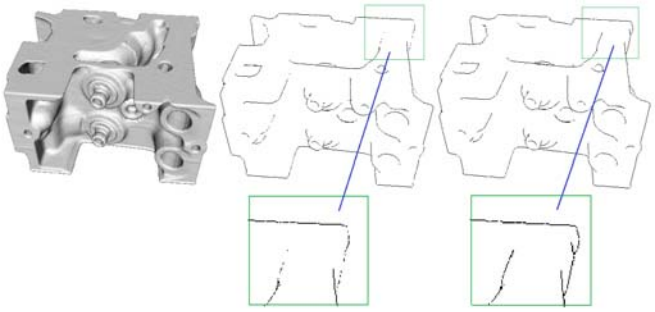

Fig. 9. Difference between rendering methods (from left to right): iso-surface rendering, silhouette rendering with ray based visibility test, silhouette rendering with geometry shader/CUDA rasterizer. The dataset is part of the Volume Library (2013).

Table 5. Percentage of correctly rendered silhouette pixels (%) for the three methods.

Data vol.	First method	Second method	Third method
a) 256^3	0.99	0.93	0.99
b) 256^3	0.97	0.91	0.97
c) 256^3	0.97	0.96	0.97
d) $128x256^2$	0.97	0.93	0.97
e) 32^3	0.97	0.90	0.97
f) 64^3	0.95	0.90	0.95

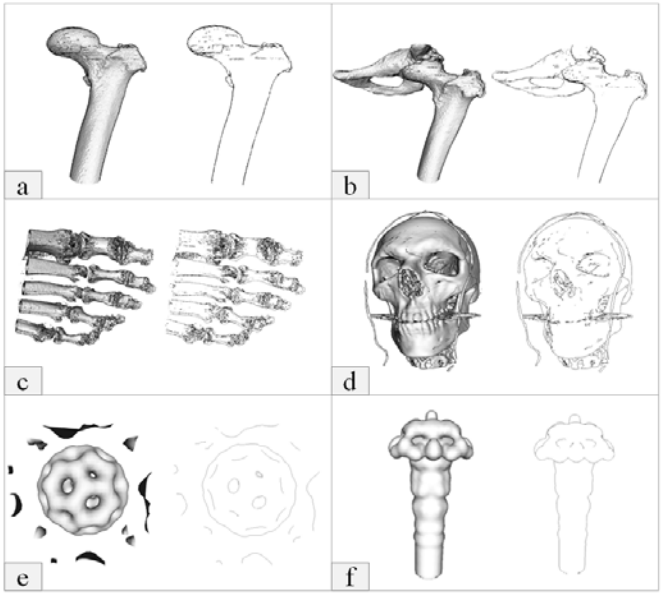

Fig. 10. Datasets used for testing the percentage of contour pixels correctly rendered. For each dataset, an iso-surface and a silhouette are shown. Datasets a) and b) represent a femoral bone and a pair of hip bones (femur+pelvis), respectively, and were obtained from Floreasca Emergency Hospital in Bucharest. Datasets c) Foot, d) Head (Visible Male), e) Bucky Ball and f) Fuel Injection are part of the Volume Library (2013).

6. CONCLUSIONS

This paper presents several contributions in the field of non-photorealistic volume rendering. Three silhouette rendering methods were implemented, each being suited for different types of datasets and different characteristics of the GPU cards.

The first implementation requires enough GPU memory to store the dataset, the reconstructed iso-surface and the extracted silhouette. The advantages of this method are the fast rendering times and the possibility to reconstruct the iso-surface only when the iso-value is changed. The silhouette rendering with marching cubes and geometry shader leads to interactive frame rates for datasets up to 512^3 if the reconstruction is done only when the iso-value is changed.

The second implementation stores on the GPU only the dataset and the silhouette of the iso-surface. The time consuming operation in the second method is the ray based visibility test, which casts rays from each silhouette vertex to the viewer's position, intersecting them with the volume. The implementation that splits the initial volume into chunks handles larger datasets, but is not very fast. The reason is that for one extracted sub-silhouette, the visibility test is run for each sub-volume, serially. Another drawback of this implementation is the requirement to extract the iso-surface and the silhouette each time the viewer's position changes.

The third silhouette rendering implementation is perfectly suited for the case when the size of the dataset makes it impossible to store the iso-surface or the silhouette on the GPU memory. Our CUDA rasterizer solves the problem of the GPU memory limitation, but it is not as efficient as the hardware rendering mechanisms of the GPU. Similar to the second implementation, the iso-surface and the silhouette have to be extracted each time the position of the viewer is changed.

ACKNOWLEDGEMENTS

Part of the work has been funded by the Sectorial Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/88/1.5/S/61178.

REFERENCES

Bruckner S., Gröller M.E. (2007). Style Transfer Functions for Illustrative Volume Rendering. *Computer Graphics Forum*, 26(3):p. 715-724.

Burns M., Klawe J., Rusnikiewicz S., Finkelstein A., DeCarlo D. (2005). Line Drawings from Volume Data. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 24(3):p. 512-518.

Co C. S., Hamann B., Joy K.I (2003). Isosplatting: A point-based alternative to isosurface visualization. *Proceedings of 11th Pacific Conference on Computer Graphics*, 325-334.

Dong F., Clapworthy G.J., Lin H., Krokos M.A. (2003). Non-photorealistic Rendering of Medical Volume Data. *IEEE Computer Graphics and Applications*, 23(4):p. 44-52.

Hadwiger M., Sigg C., Scharsach H., Buehler K., Gross M. (2005): Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):p. 303–312.

Kindlmann G., Whitaker R., Tasdizen T., Moeller T. (2003). Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications. *VIS '03 Proceedings of the 14th IEEE Visualization*, p.513-520.

Levoy M. (1990). Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):p.245-261.

Lorensen W.E., Cline H.E. (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH '87 Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* , p.163-169.

Luo Y., Guitian J.A.I., Gobbetti E., Marton F. (2009). Context Preserving Focal Probes for Exploration of Volumetric Medical Datasets. *Proceedings of 3DPH*, 187-198.

Nvidia Corporation (2011a). *NVIDIA CUDA C Programming Guide*, Version 4.0.

Nvidia Corporation (2011b). *NVIDIA CUDA SDK Example*, available at www.nvidia.com last visited in 02.01.2012.

Pelt, R.V., Vilanova, A., Wetering, H.M.M.V.D. (2008). GPU-based Particle Systems for Illustrative Volume Rendering. *Volume Graphics*, p. 89-96.

Petrescu L., Morar A., Moldoveanu F., Asavei V. (2011). Real Time Reconstruction of Volumes from Very Large Datasets using CUDA. *Proceedings of 15th International Conference on System Theory, Control and Computing* p. 1-5.

Yuan X., Chen B. (2004). Illustrating Surfaces in Volume. *Proceedings of VisSym'04 Joint IEEE/EG Symposium on Visualization.*

The Volume Library, available at http://www9.informatik.uni-erlangen.de/External/vollib/, last visited in 11.03.2013.

Morar A., Moldoveanu F., Moldeveanu A., Asavei V., Egner A. (2012). Multi-GPGPU Based Medical Image Processing in Hip Replacement", *Journal of Control Engineering and Applied Informatics*, vol. 14(3), pp. 25-34.