10.1109/ULTSYM.2009.0046

GPU-Based Reconstruction and Display for 4D Ultrasound Data

Ahmed F. Elnokrashy, Ahmed A. Elmalky, Tamer M. Hosny, Marwan Abd Ellah, Alaa Megawer, Abobakr Elsebai, Abou-Bakr M. Youssef, and Yasser M. Kadah
Biomedical Engineering Department, Cairo University and IBE Tech, Giza, Egypt
E-mail: nokrashy@ibetech.com

Abstract—**Due to the required computational effort of 4D ultrasound imaging, such systems depend on low complexity techniques like nearest neighbor interpolation, which affects volume quality. Moreover, more accurate techniques like normalized convolution, backward trilinear interpolation, and forward spherical and ellipsoidal Gaussian kernel, are avoided in real-time imaging because of the tight reconstruction time. The goal of this work is to utilize recent commercial graphics hardware technology of graphics processing unit (GPU) to speed up the reconstruction time while increasing the quality of displayed volume.**

Keywords-4D ultrasound; volume rendering; GPU; texture mapping; ray casting.

I. INTRODUCTION

Four-dimensional (4D) or real-time three-dimensional ultrasound imaging has become an important diagnostic tool for many clinical applications. Its implementation includes three consecutive steps for acquisition, volume reconstruction and visualization. Limitations of its implementation arise from its demanding computational requirements for the latter two steps that traditionally involved expensive custom processing hardware. As a result, 4D option is not currently available for most low-end and medium range ultrasound imaging systems.

Several approaches have been developed to lower the cost of 4D ultrasound systems. For example, the required computational effort of 4D ultrasound imaging can be lowered using low complexity volume reconstruction techniques such as nearest neighbor interpolation. Nevertheless, such methods affect reconstructed volume quality to a great extent. Moreover, more accurate techniques like normalized convolution, backward trilinear interpolation, and forward spherical and ellipsoidal Gaussian kernel, were not practical for real-time imaging because of the tight reconstruction time available [1].

Direct volume rendering methods compute images of a volumetric data set without explicit extraction of geometric surfaces from the data. Optical model is used to map data values to optical properties; namely, color and opacity. Then, the optical properties are projected using a suitable projection function along each viewing ray to render the data. In practical systems, volume data are stored either as a stack of 2D texture slices or as a single 3D texture object. Such data represent samples on a given sampling grid. Values in between grid elements can be computed by interpolating data at neighboring such elements. This process is known as volume reconstruction and has usually demanding computational requirements.

During the rendering process, the optical model describes how points in the volume interact with light. Optical parameters are usually taken as the data values directly, or they are derived from applying one or more transfer functions to the data values. The choice of a transfer function emphasizes desired features within the data and can be implemented as lookup tables. Output images are generated by sampling the volume along all viewing rays and accumulating the resulting optical properties to compute a pixel for each ray. Texture-based techniques perform the sampling step by rendering a set of 2D geometric primitives inside the volume.

Many real-time volume visualization techniques use texture mapping, which has several disadvantages. For example, 3D texture depends mainly on regularly sampled data on texture coordinates such as 3D rectilinear grid. Other techniques depend on regular proxy geometry to map texture. While texture mapping exploits hardware acceleration in graphics processing units (GPU), they introduce approximation artifacts in ultrasound imaging applications with polar sampling. Alternatively, ray casting visualization does not depend on proxy geometry to map data on. This alleviates approximation artifacts introduced by texture mapping. The problem with ray casting is its prohibitively high computational complexity especially in real-time applications such as 4D ultrasound. Recent GPUs support programmable pipelines and multiprocessing streaming units, which make it possible to implement ray casting in real-time using GPUs [2].

At the present, most medical volume rendering implementations are based on slice-based methods where axis- or viewport-aligned textured slices are blended together to approximate the volume rendering integral. However, slice-based implementations are rasterization-limited and difficult to optimize. Moreover, when applying such transfer functions as perspective projection, the integration step size will vary along viewing rays when using planar proxy geometries, leading to visible artifacts. As a result, slice-based techniques cannot present an optimal volume visualization framework [1].

The advent of DirectX Shader Model 3 and comparable OpenGL extensions has led to graphics processors providing an ideal platform for efficiently mapping ray casting-based volume rendering to hardware. This fragment-program based

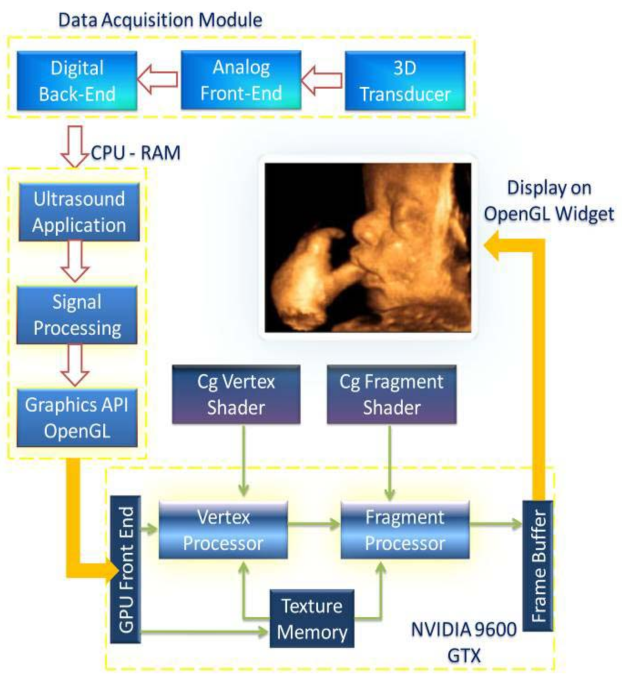

Fig. 1: Overall block diagram of the system

ray casting does not suffer from any flexibility issues and, therefore, offers an optimal volume visualizing tool. Moreover, new graphics hardware designs encourage the use of novel fragment program features with much faster current and planned generations of graphics hardware. Hence, our goal in this work is to take advantage of this new direction and utilize recent commercial graphics hardware technology of GPUs to speed up the volume reconstruction time while increasing the quality of visualization. The target is to achieve high-quality volume rendering using ray casting at frame rates suitable for present 4D ultrasound imaging applications.

II. GPU BASICS

Over the past decade, GPUs have become widely available in PCs. GPUs feature powerful computing resources and high memory bandwidth, which enable high performance on 3D graphics applications. 3D graphics computations are organized into a graphics pipeline that outlines the series of computation stages between the scene input and the image output. The input to the graphics pipeline is a scene consisting of a list of geometry (defined as connected vertices) and a set of graphics instructions to compute the scene from the geometry. The GPU then processes and maps those vertices into screen-space geometry, which in turn is divided into pixel-sized fragments, in a process called *rasterization*. Each fragment is then associated with a pixel position on the screen. Finally, the fragments are processed and assembled into an image made of pixels [3].

The pipeline contains on the order of a dozen stages, each of which is implemented a separate processor per stage. The typical input to the pipeline is tens to hundreds of thousands of vertices, each of which can be processed in parallel on the GPU. The complex graphics pipeline is thus divided in space, with separate processors on the GPU running each stage in parallel. This is different from that of a CPU, which only features a single processor [3].

From a programming perspective, GPUs feature two programmable stages; namely vertex and fragment processors. The vertex stage runs a user-defined program on each input vertex input, while the fragment stage runs a different program on every fragment. Both stages have a similar programming model and are most efficient when run on long lists of inputs independently. That is, many vertices or fragments can be processed in parallel under SIMD (single-instruction, multiple-data) control, meaning that each vertex or fragment is computed in parallel with the same sequence of instructions controlling each computation. Moreover, that such programs can read from any location in global memory but cannot write to arbitrary global memory. Instead, the output from a vertex program is a single vertex, and the output from a fragment program is just one fragment at the fragment's pixel position. The aggregate arithmetic rate of the fragment programs in Nvidia's GeForce GPUs for example is more than 100 billion floating-point operations per second. while simple applications can be expressed in a single pass of the graphics pipeline, complex ones may use multiple passes through the graphics pipeline by using the global image output of one pass in the computation of subsequent passes. Multiple GPU cards can be installed in parallel to handle larger sizes of data as well. With its superior computation rat and low cost, the GPU has the potential to be the computational powerhouse piece in medical imaging application such as 4D ultrasound imaging [3].

III. METHODS

Fig. 1 shows the overall block diagram of the system where the data are acquired under and fed into the graphics card to handle the reconstruction and visualization of the volume. The basic steps of the software are shown in Fig. 2 and will be described in detail as follows.

A. *Volume Data Representation*

At the start, volume data have to be stored in memory in a suitable format after either coming from the digital backend of the ultrasound machine or streamed from a stored sample on the hard disk drive. This is considered as a preparation step to make the data ready in a suitable form to be downloaded to the graphics hardware as textures. Depending on the kind of proxy geometry used, the volume can be either stored in a single block, when view-aligned slices together with a single 3D texture are used, or split up into three stacks of 2D slices, when object-aligned slices together with multiple 2D textures are used. Usually, it is more convenient to store the volume only in a single 3D array, which can be downloaded as a single 3D texture, and extract data for 2D textures on-the-fly, just as needed. This requires having a graphics hardware that supports 3D texture mapping. Depending on the complexity of the rendering mode, classification and illumination, there may even be several volumes containing all the information needed. Likewise, the actual storage format of voxels depends on the rendering mode and the type of volume, e.g., densities, gradients, gradient magnitudes, etc. Conceptually different volumes may also be combined into the same actual volume such as combining gradient data and density data in RGBA voxels. Although data representation is usually a part of preprocessing, it is not necessarily so. New data may have to be generated on-the-fly when the rendering mode or specific parameters are changed. This component is usually executed only once at startup or only executed when the rendering mode changes.

B. *Transfer Function Initialization*

Transfer functions are usually represented by color lookup tables and can have multiple dimensions stored as simple multidimensional arrays. This component is usually designed to be user-selectable, to provide the flexibility to change the transfer function from the user interface depending on the application.

C. *Color Map Initialization*

The color map is prepared from the four transfer functions that represent the color components red, green, blue, and the opacity transfer function. After setting these transfer functions, color maps must be packed to be then invoked into the GPU as a 1D texture map to be fetched and used in the fragment program in order to add the color values to the scene being rendered.

D. *Fragment Shader configuration*

There are two ways to terminate a viewing ray. The regular termination takes place once the ray leaves the volume, and is calculated by comparing the travelled distance to the length of the original viewing vector stored in the direction texture. However, to do this in one pass, the GPU has to be able to execute conditional breaks inside the loop, which requires a Shader Model 3 capable graphics card. The introduced early ray termination can also only be efficiently implemented on such a GPU, since it requires one additional condition after every single sample, checking whether the accumulated alpha values have exceeded a certain threshold. By using the conditional registers introduced with the newest generation of graphics cards, these two checks can be carried out together, resulting in only one conditional break statement. Implementing this with Shader Model 2 would require a separate ray termination pass, where we face a trade-off between two techniques. The first has a termination pass after every sample and hence requires 2 passes per sample, but provides the ability to exactly terminate the ray where necessary thus only calculating samples that are part of the final image. On the other hand, the second executes the termination pass only after a number of ray-casting passes, which could themselves again calculate a number of samples, and hence has a lower impact on performance. It however introduces a problem whereby the ray may be sampled outside the bounding box. For a simple bounding box setup, this may not be a major disadvantage because the volume outside the bounding box is empty anyway, leading only to a small performance hit. Alternatively, when introducing advanced techniques like cache textures or geometry intersection, it often has to be ensured that rays are terminated correctly, because

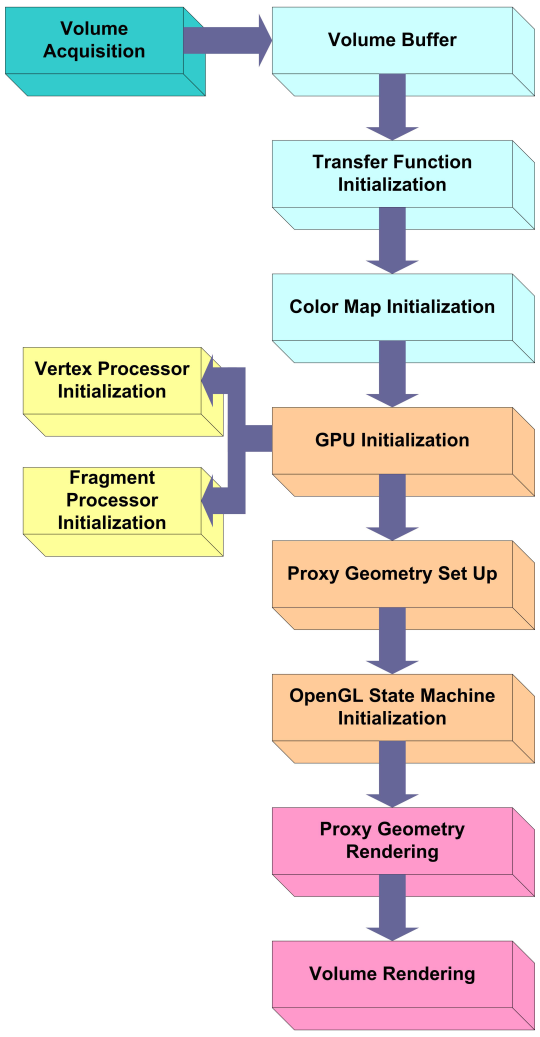

Fig. 2: Block diagram of the software

samples after the termination position may already be invalid or at least should not be part of the final image. Thus, it makes sense to restrict the system requirements to Shader Model 3 enabled graphics cards to account for all future enhancements of the algorithm and keep the pipeline as flexible as possible.

E. *Volume textures*

In order for the graphics hardware to be able to access all the required volume information, the volume data must be downloaded and stored in textures. At this stage, a translation from data format (external texture format) to texture format (internal texture format) might take place, if the two are not identical. Usually and in many datasets we have working with this is the case, so the conversion or translation from data format to texture format must take place. This component is usually executed at startup, when the rendering mode changes, or when a new 3D data set is ready to be rendered. How and what textures containing the actual volume data have to be downloaded to the graphics hardware depends on a number of factors, most of all the rendering mode and type of classification. Proxy Geometry Rendering The last component of the execution sequence outlined in this section is getting the

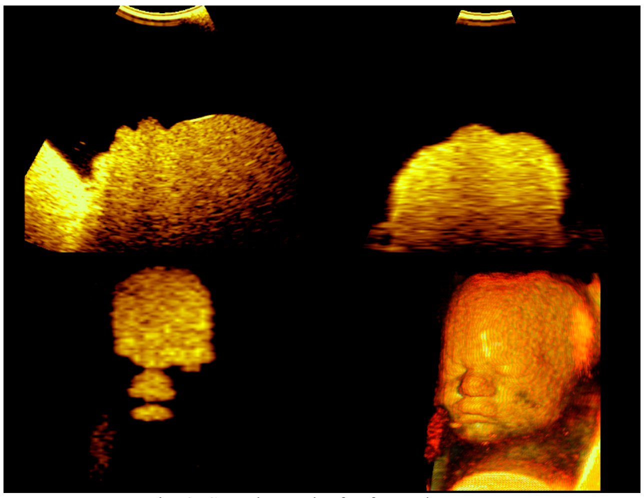
Fig. 3: Sample results for fetus phantom

graphics hardware to render geometry. This is what actually causes the generation of fragments to be shaded and blended into the frame buffer, after resampling the volume data accordingly.

IV. Results and Discussion

Test data were acquired using 3D mechanical transducer (Prosonic, inc.) interfaced to IBE Tech Sonata ultrasound imaging system using a special research console designed for this project. Volume size was 256×256×225, fan angle of 86 deg, and scan depth of 160 mm. The implementation of the proposed method was done using Visual Studio C++, OpenGL and Cg (NVIDIA, inc.). A personal computer with Intel Core 2 Quad CPU with 8 GB RAM, and an NVIDIA GeForce 9600 graphics card with GPU was used under Windows XP operating system. The system was used to scan the CIRS 36-week Fetal Ultrasound Training Phantom Model 065-36 (CIRS, inc.). Sample output from the developed system are shown in Fig. 3 where three perpendicular sections are displayed in addition to the rendered volume in real-time.

Due to the limitations in data acquisition imposed by the scanner, it was not directly possible to measure the maximum frame rate of the volume rendering part. In order to do that, the system was allowed to update volume data and perform volume rendering without having to wait for a complete volume to be acquired. This means that the acquisition and reconstruction parts are made independent. In this case, the volume rendering frame rate was found to vary between 24 frames/s and 85 frames/s with an average of about 60 frames/s. The frame rate was found to depend on the view whereby the frame rate is at minimum when the probe fanning is in the front direction of the view. This is due to the fact that the required interpolation calculations depend on the view and in this particular view the interpolation required is at maximum. In all cases, the frame rates obtained are sufficient for practical 4D ultrasound and it appears that the bottleneck in this technology is on the acquisition side not the volume rendering side.

V. Conclusions

A new low-cost 4D ultrasound reconstruction and display system is introduced. The new system takes advantage of the recent advances in computer graphics hardware and in particular the GPU technology to reconstruct high quality volumes at high frame rates. The system has potential to make 4D ultrasound more affordable for low to medium range ultrasound imaging systems.

References

[1] S. Stegmaier, M. Strengert. T. Klein, and T. Ertl, “A simple and flexible volume rendering framework for graphics-hardware-based raycasting,” *Proc. Fourth International Workshop on Volume Graphics*, June 2005.

[2] T. Sumanaweera, “Applying real-time shading to 3D ultrasound visualization,” in *GPU Gems*, R. Fernando, Ed., Addison-Wesley, New York, pp. 693-707, 2004.

[3] J.D. Owens, “GPUs tapped for general computing,” EETimes.com, article ID: 55300884, 2004.