Accelerated Ray Casting Framework for Dynamic Volume Dataset

***[*] ***[†]
******* *******

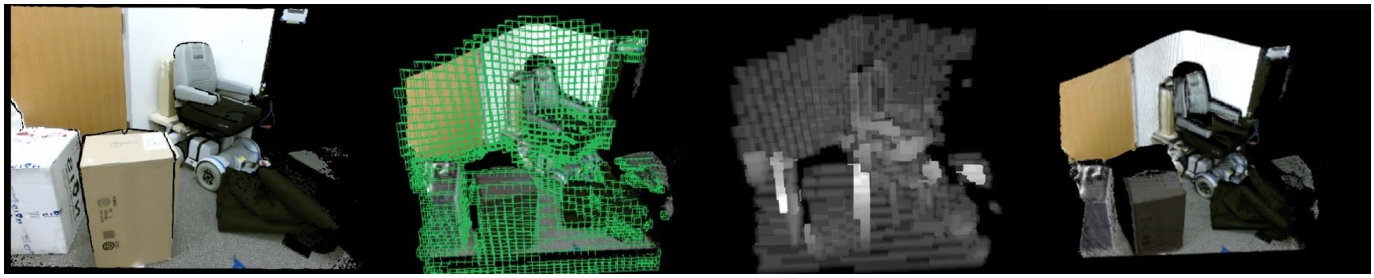

Figure 1: *Dynamic TSDF Volume Visualization Pipeline: the images from left to right are: 1. Registered RGB-Depth image from the Kinect; this image is used to update the TSDF volume (normally at 30Hz); 2. Visualization of our TSDF volume overlayed with real-time generated visible bricks; 3. Visualization of the Start End Position texture showing the traveling distance each ray actually will go through, which is the output of the left image; 4 Final rendering of the dynamic TSDF.*

Abstract

We present a flexible and extensible accelerated ray casting framework for dynamic volume data. Our target application is the real time reconstruction and rendering of dynamic 3D scenes based on streaming data from RGB+D sensors. For such data, traditional methods that rely on substantial preprocessing are impractical. Our framework derives its efficiency from a novel two-stage approach that couples metadata reconstruction with volume update to hide the overhead. In benchmark tests, we find that our approach enables up to a 2X improvement in rendering speed for dynamic volumes compared to a traditional raycasting pipeline.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture I.3.6 [Computer Graphics]: Methodology and Techniques—Ray Casting;

Keywords: Computer Graphics, GPU, Volume Rendering, Volume Visualization, Ray-Casting, Dynamic Volume

1 Introduction

Volume rendering [Levoy 1988] [Drebin et al. 1988] encompasses a wide range of techniques for generating 2D images from volumetric datasets such as MRI, CT, etc. Motivated by the importance of volume rendering in a wide range of application areas, such as medical science, fluid dynamics, geology, meteorology, and abstract mathematics, many researchers have devoted considerable efforts over the past decades to advancing the state-of-the-art in rendering quality and performance.

[*]e-mail:***@***.***
[†]e-mail:l**@***.***

Generally speaking, volume rendering methods can be divided into two basic categories: indirect volume rendering and direct volume rendering. In this paper we focus on Direct Volume Rendering, in which 2D images are rendered directly from volumetric data without an intermediate surface extraction step, by evaluating an optical model that describes the interaction of light with the volumetric data [Max 1995].

While the original volume rendering algorithms were non-interactive and focused on the rendering of individual images, advances in graphics hardware soon led to the development of methods by which volume rendering could be achieved at interactive frame rates. The most popular of these early methods was texture-based volume rendering, which was implemented by sampling the volume data using a stack of proxy geometry and then blending the resulting textured slices in a way that approximated the integration of light as it traveled through the volume [Cullip and Neumann 1993] [Cabral et al. 1994].

The late 1990s to the early 2000s saw significant advances in the functionality of the texture-based method as researchers took advantage of the new capabilities introduced by modern GPUs. [Engel et al. 2001] proposed pre-integrated volume rendering, which greatly reduced artifacts caused by the slice-based sampling, and [Kniss et al. 2002] introduced the use of multidimensional transfer functions. [Li et al. 2003] adapted ray casting acceleration techniques, such as early ray termination and empty space skipping, to the texture-based method. The texture slice-based method became a standard for interactive volume rendering.

At the same time, researchers were still investigating the possibility of using the GPU to do ray-casting-based volume rendering, since the slice based method had some inherent disadvantages, such as reduced rendering quality, imbalanced rasterization overhead [Kruger and Westermann 2003], and the difficulty of implementing elaborate optical models [Stegmaier et al. 2005]. [Parker et al. 1999] developed one of the first systems capable of achieving real time volume rendering using a ray tracing approach, taking advantage of the computing capabilities of a large shared-memory multiprocessor. [Roettger et al. 2003] were among the first researchers to present a GPU-based raycasting algorithm implementation, which was able to achieve higher quality images at near-interactive rates of 1-3 seconds per frame on consumer-level hardware, and [Stegmaier et al. 2005]presented the first single pass GPU-based ray casting al-

gorithm, leveraging new features of the GPU; final rendering rates were still only half of the slice based method however.

Later, as the GPU became much more powerful, the performance of GPU-based ray casting methods began to match that of the slice-based methods. At that point, the higher rendering quality and greater flexibility of the ray-casting approach gave it an overwhelming advantage, and ray-casting again became the state-of-the-art technique for interactive volume rendering [Hadwiger et al. 2008].

During recent few years, GPU-based ray casting algorithms have been well-studied over the recent years. CUDA based GPGPU implementations [Marsalek et al. 2008] have been reported to be even faster than previous GPU implementations, and many people have been working on improving GPU ray casting algorithms with the ability to render larger volume datasets with better quality [Gobbetti et al. 2008].

Nowadays, with the explosive growth in data, many volumetric datasets can no longer fit into GPU memory or even main memory. This led researchers to focus on the development of out-of-core methods, which can now handle up to tera- or even petascale volumes. The efficiency of these renderers relies on their memory management, and many efficient approaches have been proposed, such as: ray-guided approaches [Crassin et al. 2009] [Fogal et al. 2013], and visualization-driven rendering [Hadwiger et al. 2012]. For a detailed review of recent research in large-scale volume rendering, please refer to the recent survey by [Beyer et al. 2014].

With the evolution of todays powerful GPU, and the continued development of advanced volume rendering techniques, it is becoming increasingly feasible to use volumetric data for interactive special effects simulation and even visual art. Many simulations and visual effects involve materials that are volumetric in nature (e.g. fluid, fire, smoke, fog) and problematic to effectively process and render using 1D or 2D data structures. However, this type of application introduces the requirement of working with data in which the value stored in any given voxel can change at any time, either due to the procedural evolution of the simulation or to user-initiated actions. In order to render a dynamically changing volume at interactive frame rates, the need for ultra-fast rendering performance becomes urgent, since merely updating the volume data at every frame will already consume a lot of computational power.

With respect to our driving application, where we need to achieve fast surface reconstruction from a relatively low-resolution (e.g. 512^3) live stream of volumetric data representing a truncated signed distance field (TSDF), all of the previous approaches mentioned above cannot satisfy our requirements in terms of performance. In this paper, we present a novel accelerated ray casting framework that we developed to address this issue.

2 Related work

In this section, we introduce some of the existing acceleration techniques we considered for our application, and review other work closely related to the visualization of dynamic volume data.

2.1 Techniques

Early Ray Termination(ERT: First introduced by [Levoy 1990] is an acceleration technique that achieves greater efficiency by reducing the consideration of unnecessary samples. ERT is motivated by the fact that the accumulated intensity value for a cast ray will not noticeably change once the opacity along that ray has reached a certain threshold level. Due to its simplicity and effectiveness, this technique is widely used in volume rendering when the volume is rendered in a front-to-back or back-to-front manner. We adopted this technique in our system to avoid unnecessary volume access, and in a GPU implementation, the only overhead involved in implementing ERT is a single conditional evaluation per iteration, which is negligible.

Empty Space Skipping(ESS): In volumetric rendering, a common observation is that after classification (by means of transfer function), only some parts of the entire volume will be visible in the final result; large parts of the volume may be transparent, and have no impact on the final image. Thus, in standard ray casting, a huge amount of processing time can be wasted on sampling transparent voxels. In the case of iso-surface rendering, the situation is even worse: for each ray, along their sampling path, only 2 or 3 samples will contribute to the final pixel color. Long ago, researchers realized that this waste of processing time and I/O bandwidth could be nicely avoided with almost no runtime overhead [Levoy 1990] through the use of a CPU preprocessing pass, in which transparent regions were flagged to be skipped during rendering. This method became the most efficient and widely used empty space skipping technique. However, the need for costly preprocessing makes it impractical for use with dynamic data. In our method, we address this issue by making use of the volume updating phase and the geometry shader stage. We will explain in more detail about this in 3.3.

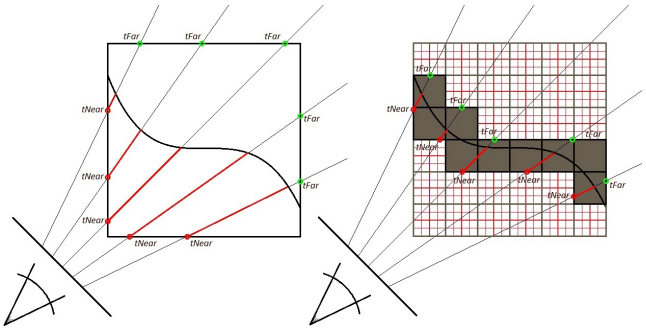

Figure 2: *This figure shows the differences between normal isosurface ray casting (left) and Empty Space Skipping iso-surface ray casting (right). Each image shows a view-direction-aligned slice in volume ray casting, where the projection plane is located at the left bottom corner, the black big rectangle is the boundary of the volume data, and the curved line segment is the iso-surface. The small red grid in the right image represents the actual data voxels, and the medium sized grey grid represents bricks, which contain 16 2D data voxels. Bricks shown with a brown interior contain non-transparent voxels, indicating that all voxels inside these bricks will be sampled; all others will be skipped during ray casting since they are inactive or make no contribution to the final image value. In the left image, the red portion of each ray line shows the difference between these two methods in terms of sample count (performance).*

Space Leaping: Space leaping is a more general ESS acceleration technique for ray casting [Yagel and Shi 1993] [Freund and Sloan 1997]. The way this method works is by advancing the ray directly to the next interesting sample point rather than simply incrementing the ray by a pre-defined small step. An auxiliary volume is used to store the information needed to do the skipping. Space leaping therefore targets the same problem as ESS the waste of computational iterations and I/O bandwidth on transparent voxels but it can be implemented with much more flexibility. An example of a space leaping technique that is different from EES is proximity clouds [Cohen and Sheffer 1994]], in which the distance to the nearest interesting region is stored in the empty space voxels. However, the proximity cloud approach requires a much more expensive preprocessing pass than EES (as well as other efficient space leaping methods), so we decided not include this technique into our system.

Min-Max Encoding Octree: Strictly speaking, a Min-Max Encoding Octree, first introduced by [Kruger and Westermann 2003], is an enhanced version of the empty space skipping technique. It enables efficient data access even when the transfer function changes. This method also requires a preprocessing step in which information is stored to indicates the regions to be skipped. Instead of flagging transparent regions directly, however, this method stores the max and min values of each region. During rendering, the current visibility of each encoded region is determined by evaluating the transfer function and comparing the resulting density with the stored min-max threshold values to determine whether this region is transparent or not, and skipping or rendering this region accordingly. The pre-processing pass for constructing a min-max encoding octree is a little bit more expensive than normal EES, and slightly more memory is required to store the additional data. So, for efficiency considerations we didnt adopt this approach in our framework.

Deferred Shading: Deferred shading, first introduced by [Deering et al. 1988] is an efficient image-space shading technique. The key concept of this method is to delay expensive operations (such as complex shading) to a later processing stage where they will only be applied to pixels that actually need to be processed. The deferred rendering pipeline allows efficient rendering of iso-surfaces [Hadwiger et al. 2005] and using expensive transfer functions wont badly hurt the performance. The disadvantage of deferred shading is its high memory and bandwidth cost. Also, it cannot be applied in situations that require accumulative rendering. Due to this, deferred shading currently is out of our options.

2.2 Related Work in Dynamic Volume Rendering

A few works talk about visualizing dynamic volume data. [Bruder et al. 2011] present a poster in which the Voreen system is used to render real-time 4D ultrasound data. [Elnokrashy et al. 2009] present a basic pipeline for GPU-based ultrasound rendering, but their pipeline relies on simple ray casting with no acceleration technique applied. The work most related to our system is by [Soltteszova et al.], where they present a visibility-driven processing and visualization system for streaming volume data (a live 3D ultrasound volume). The volume rendering method they adopted can be counted as the first accelerated ray caster for dynamic volume visualization, but the key for their fast ray casting heavily depends on a by-product of their proposed filtering method, which is costly and may not be applicable in general purpose ray casting.

3 Accelerated Ray Casting Framework

3.1 Background

The most well-known advanced ray casting pipeline, well summarized by [Hadwiger et al. 2008] , mainly exploits two acceleration schemes: empty space skipping and early ray termination. The brief workflow of this pipeline can be depicted as in figure 3.

This pipeline works as follows (please refer to figure 3):

1. In a CPU pre-pass, a regular grid of bricks is constructed with a much lower resolution than the original volume data (normally a brick contains 8^3 data voxels). Then the min-max voxel value inside each brick is stored by iterating through the volume one time. (CPU pre-pass in figure 3)

2. Inactive bricks are culled on the CPU by comparing the min-max value of each brick with the iso value (for iso-surface ray casting) or opacity given by the transfer function (for normal ray casting). A bit array is generated to flag active bricks. (CPU Pass in figure 3)

3. Rasterize the active bricks boundary faces normally into the depth buffer, to get the information for the ray start position. (GPU Pass 1 in figure 3)

4. Rasterize the active bricks boundary faces with front face culling and reverse the depth test to get the information for the ray end position. (GPU Pass 2 in figure 3)

5. Use the ray start and end information from previous passes to guide the ray casting with early ray termination. (GPU Final Pass in figure 3)

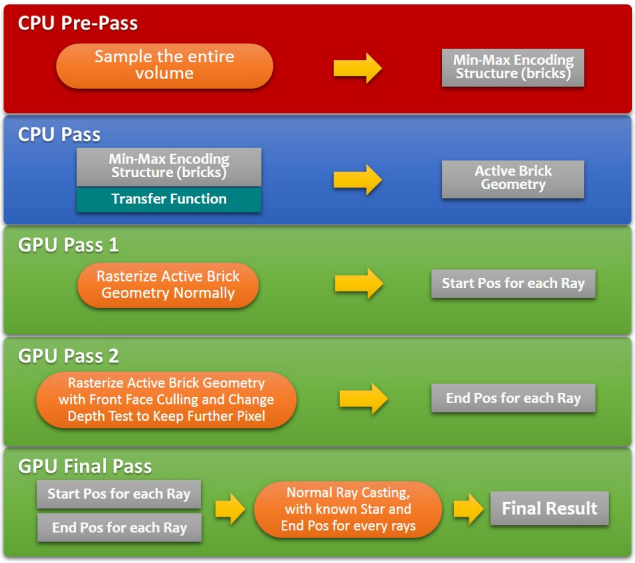

Figure 3: *The ray casting pipeline summarized by HADWIGER*

Our approach generally follows this standard design framework. However, naively applying this pipeline directly to the rendering of dynamic volume data will not work. The speedup realized by [Hadwiger et al. 2008] is heavily dependent on the Min-Max Encoding structure, which takes lots of time to construct. Furthermore, the encoding has to be re-computed whenever any of the original data values inside the volume change. This overhead will quickly become prohibitive if we need to redo the CPU pre-pass at every frame.

Our solution to the aforementioned problem is to utilize the GPU to do the preprocessing; and, specifically, to integrate the preprocessing with the volume updating phase. To further reduce the overhead of inactive brick culling, we use a geometry shader for fast GPU culling, and we use alpha blending to generate the ray start texture and ray end texture simultaneously. Details are explained in the following section.

3.2 Terminology

Before getting further into the details of our implementation, we define some terms:

data volume: the 3 dimensional uniform grid of data to be visualized; the 3D texture of original data;

voxel: a single element in the data volume, which can hold a scalar or vector value;

flag volume: the 3 dimensional uniform grid of encoded data for

accelerated ray casting, used to annotate regions in the data volume. The flag volume will span the same physical size as the data volume, but will only have 1/4 ,1/8 or 1/16 the resolution in each dimension;

brick: a single element in the flag volume, mapping to 4^3, 8^3 or 16^3 data voxels. Thus, each brick in the flag volume corresponds to a region in the data volume.

3.3 Implementation

Our framework uses one additional flag volume to mark transparent regions. The procedure is a two stage process:

1. (Updating Phase)Flag Volume Reset: right before volume updating, the flag volume needs to be reset. This process uses a separate compute shader pass to clean up the whole flag volume. Resetting the whole volume inside GPU memory has little overhead, and the memory size is much small than the data volume (normally only $(1/8)^3$ * (flagvolume-databit)/(datavolumedatabit)).

2. (Updating Phase)Flag Volume Update: during the volume update, each updated voxel will map its new value to opacity through the transfer function, and set the corresponding brick in the flag volume if its not transparent. In this stage, the resolution of the flag volume resolution may have an impact on performance: low flag volume resolution results in in too many voxels being mapped to one brick, which could cause GPU memory bank conflict. We have had good results using a flag volume that is $(1/8)^3$ the size of the data volume.

3. (Rendering Phase)Generate Ray Start-End Texture: Rasterize the mesh for non-transparent bricks and output the depth data using both the color channel and the alpha channel with specified alpha blending settings to keep the smallest value in the color channel and the largest value in the alpha channel. The resulting image will be a 4-channel depth map with the color channel storing the smallest depth and the alpha channel storing the largest depth. In this stage, instead of using the CPU to generate the mesh for active bricks, we use the geometry shader to generate active bricks in the GPU directly, to reduce the potential bottleneck introduced by CPU-GPU communication. The use of slightly-expensive alpha blending instead of light-weight depth testing, is compensated by the need to use only one pass to generate both the near and far depth values, since the old way of doing two passes of intermediate rendering with a customized geometry shader may introduce more overhead.

4. (Rendering Phase)Final Ray Casting: With the near-far depth texture from the previous pass, ESS ray casting can be implemented using any optical model and shading techniques.

To summarize, our dynamic volume ray casting framework utilizes the volume updating stage which enables realtime encoding structure updating to bring efficient empty space skipping technique into the dynamic volume ray casting algorithm. Figure 6 provides a visual overview of the system structure.

To summarize, our dynamic volume ray casting framework enables empty space skipping to be efficiently applied to dynamic volume data through the introduction of a volume updating stage on the GPU, wherein the encoding structure for empty space skipping is updated in real time as the volume changes. Figure 6 provides a visual overview of the system structure.

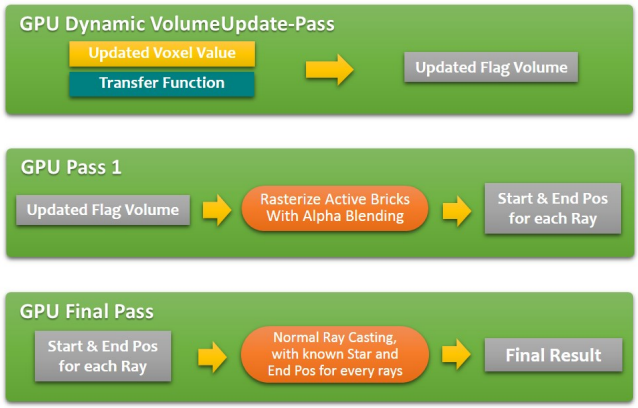

Figure 6: *The overview of our accelerated ray casting framework.*

4 Result and Discussion

Due to the lack of a well-known dynamic volume dataset, we have created our own data to demonstrate the effectiveness of our proposed framework. The data volumes we created represent two different kinds of typical dynamic volumetric dataset used in real-time simulation and surface fusion: a dynamic density volume and a live TSDF volume.

The two datasets are ***Metaballs***: a simulated density field in which 20 radiative density centers move in an orbit around the volume center with different orbit radii, phase and speed, as shown in figure 7; and ***Live TSDF***: Truncated Signed Distance Field in which each voxel stores the truncated distance to the closest surface in the volume. In our test, we use a Kinect's RGB and Depth live streams to update the TSDF volume along with a color volume in real-time without doing actual surface fusion; see figure 4 and [Newcombe et al. 2011] for detail.

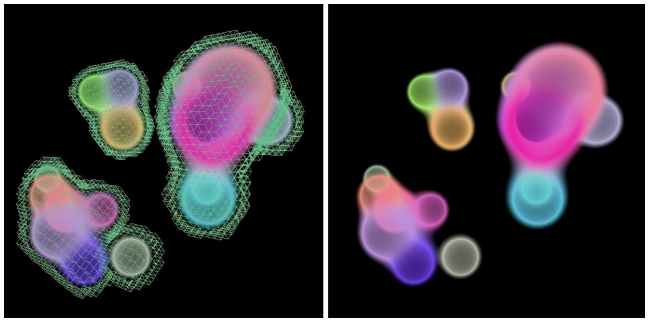

Figure 7: *Metaball dataset, a dynamic density field with 20 changing density functions. A compute shader keeps updating this volume at every frame*

In figure 5, we give exact timings in terms of CPU and GPU time for our framework, which are compared to times given by the standard ray casting pipeline mentioned in subsection 3.1, with the exception that empty space skipping is unavailable. For each dataset, we render the volume at two different resolutions to check the scalability of these methods under different data size. Also both iso-surface ray casting and the accumulative ray casting are performed on the Metaball dataset to see how the rendering method can have impact the performance.

The result clearly shows that our proposed framework brings the huge speedup that comes from empty space skipping to dynamic

Figure 4: *Dynamic TSDF Volume: the left image is a registered RGB-Depth image from the Kinect. This image is used to update the TSDF volume (normally at 30Hz). The image in the middle is a visualization of the Start End Position texture showing the traveling distance each ray actually will go through, which is the output of the right image. The right image is a visualization of the active bricks overlay onto the final image after ray casting.*

Dataset And Format	Resolution	Standard Ray Casting			Our Method		
		Simulation(μs)	Rendering(μs)	Overall	*Simulation*(μs)	Rendering(μs)	Overall
MetaBall (isosurface) R32G32B32A32	256*256*256	*1484(CPU) 17811(GPU)*	*87(CPU) 16370(GPU)*	*37*fps*	*2027(CPU) 18057(GPU)*	*146(CPU) 1086(GPU)*	*85*fps*
	384*384*384	*2550(CPU) 60395(GPU)*	*191(CPU) 25015(GPU)*	*17*fps*	*2417(CPU) 60002(GPU)*	*460(CPU) 1559(GPU)*	*27*fps*
MetaBall (accumulative) R32G32B32A32	256*256*256	*1492(CPU) 18005(GPU)*	*186(CPU) 23060(GPU)*	*30*fps*	*767(CPU) 17945(GPU)*	*112(CPU) 2263(GPU)*	*78*fps*
	384*384*384	*2419(CPU) 60481(GPU)*	*171(CPU) 36704(GPU)*	*14*fps*	*1319(CPU) 60081(GPU)*	*1709(CPU) 3324(GPU)*	*27*fps*
TSDF from Kinect R16G16 & R10G10B10A2	384*384*384	*1129(CPU) 7116(GPU)*	*702(CPU) 10769(GPU)*	*51*fps*	*6139(CPU) 7728(GPU)*	*1776(CPU) 5393(GPU)*	*70*fps*
	512*512*512	*1291(CPU) 16904(GPU)*	*912(CPU) 15099(GPU)*	*23*fps*	*1614(CPU) 18056(GPU)*	*137(CPU) 7326(GPU)*	*40*fps*

Figure 5: *The overall framerate is captured without enabling profile tools, so data may not be consistent with other timing measurements. The volume for Metaballs dataset, is a 4 channel 3D texture with 32bit for each channel. TSDF dataset have two separated volumes, one for density and weight (a 2 channel volume with 16bit/channel), and one for color(a 4 channel volume with 10bit for red, 10bit for green, 10bit for blue and 2bit for alpha). All tests have been performed on Alienware M18x: Intel Core i7-3820QM @ 2.70GHz CPU, 8GB ram, Nvidia 680M graphics card with 2GB video memory.*

volume ray casting, and the overhead of constructing encoding data is negligible.

5 Conclusion and Future work

We have presented an efficient framework for visualizing dynamic volume data. Our system is based on the standard ray casting pipeline, but by using the GPU to maintain a flag volume during the volume update phase, we allow empty space skipping to be efficiently applied to dynamic volume data. Also, by using the geometry shader along with alpha blending, we reduce the total overhead to a minimum. With these together, our system achieved great speedup for dynamic volume rendering which could not be made before. The idea of coupling rendering with volume updating also makes lots of other techniques available for dynamic volume rendering, especially those that require a pre-pass to build additional data structures. We are now exploring ways to integrate more flexible encoding volumes (e.g. min-max volumes instead of flag volumes) to enable more advanced rendering while maintain interactive framerates.

References

BEYER, J., HADWIGER, M., AND PFISTER, H. 2014. A survey of gpu-based large-scale volume visualization.

BRUDER, R., JAUER, P., ERNST, F., RICHTER, L., AND SCHWEIKARD, A. 2011. Real-time 4d ultrasound visualization with the voreen framework. In *ACM SIGGRAPH 2011 Posters*, ACM, 74.

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, ACM, 91–98.

COHEN, D., AND SHEFFER, Z. 1994. Proximity cloudsan acceleration technique for 3d grid traversal. *The Visual Computer 11*, 1, 27–38.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 15–22.

CULLIP, T. J., AND NEUMANN, U. 1993. Accelerating volume reconstruction with 3d texture hardware.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, vol. 22, ACM, 21–30.

DREBIN, R. A., CARPENTER, L., AND HANRAHAN, P. 1988. Volume rendering. In *ACM Siggraph Computer Graphics*, vol. 22, ACM, 65–74.

ELNOKRASHY, A. F., ELMALKY, A. A., HOSNY, T. M., ELLAH, M. A., MEGAWER, A., ELSEBAI, A., YOUSSEF, A., AND KADAH, Y. M. 2009. Gpu-based reconstruction and display for 4d ultrasound data. In *Ultrasonics Symposium (IUS), 2009 IEEE International*, IEEE, 189–192.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, 9–16.

FOGAL, T., SCHIEWE, A., AND KRUGER, J. 2013. An analysis of scalable gpu-based ray-guided volume rendering. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, IEEE, 43–51.

FREUND, J., AND SLOAN, K. 1997. Accelerated volume rendering using homogeneous region encoding. In *Proceedings of the 8th conference on Visualization'97*, IEEE Computer Society Press, 191–ff.

GOBBETTI, E., MARTON, F., AND GUITIÁN, J. A. I. 2008. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9, 797–806.

HADWIGER, M., SIGG, C., SCHARSACH, H., BÜHLER, K., AND GROSS, M. 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Computer Graphics Forum*, vol. 24, Wiley Online Library, 303–312.

HADWIGER, M., LJUNG, P., SALAMA, C. R., AND ROPINSKI, T. 2008. Advanced illumination techniques for gpu volume raycasting. In *ACM Siggraph Asia 2008 Courses*, ACM, 1.

HADWIGER, M., BEYER, J., JEONG, W.-K., AND PFISTER, H. 2012. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *Visualization and Computer Graphics, IEEE Transactions on 18*, 12, 2285–2294.

KNISS, J., KINDLMANN, G., AND HANSEN, C. 2002. Multidimensional transfer functions for interactive volume rendering. *Visualization and Computer Graphics, IEEE Transactions on 8*, 3, 270–285.

KRUGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, IEEE Computer Society, 38.

LEVOY, M. 1988. Display of surfaces from volume data. *Computer Graphics and Applications, IEEE 8*, 3, 29–37.

LEVOY, M. 1990. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG) 9*, 3, 245–261.

LI, W., MUELLER, K., AND KAUFMAN, A. 2003. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, IEEE Computer Society, 42.

MARSALEK, L., HAUBER, A., AND SLUSALLEK, P. 2008. High-speed volume ray casting with cuda. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, IEEE, 185–185.

MAX, N. 1995. Optical models for direct volume rendering. *Visualization and Computer Graphics, IEEE Transactions on 1*, 2, 99–108.

NEWCOMBE, R. A., IZADI, S., HILLIGES, O., MOLYNEAUX, D., KIM, D., DAVISON, A. J., KOHI, P., SHOTTON, J., HODGES, S., AND FITZGIBBON, A. 2011. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, IEEE, 127–136.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *Visualization and Computer Graphics, IEEE Transactions on 5*, 3, 238–250.

ROETTGER, S., GUTHE, S., WEISKOPF, D., ERTL, T., AND STRASSER, W. 2003. Smart hardware-accelerated volume rendering. In *VisSym*, vol. 3, Citeseer, 231–238.

SOLTESZOVA, V., VIOLA, Å. B. I., AND BRUCKNER, S. Visibility-driven processing of streaming volume data.

STEGMAIER, S., STRENGERT, M., KLEIN, T., AND ERTL, T. 2005. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics, 2005. Fourth International Workshop on*, IEEE, 187–241.

YAGEL, R., AND SHI, Z. 1993. Accelerating volume animation by space-leaping. In *Visualization, 1993. Visualization'93, Proceedings., IEEE Conference on*, IEEE, 62–69.