Volume Graphics (2006),
T. Möller, R. Machiraju, T. Ertl, M. Chen (Editors)

SIMD-Aware Ray-Casting

Warren Leung Neophytos Neophytou Klaus Mueller

Center for Visual Computing, Computer Science, Stony Brook University

Abstract

With the addition of loop and branch capabilities into the GPU programming set, a more natural and free-flowing volume rendering pipeline execution mode recently emerged. While the execution of the looped fragment program is still SIMD, the numerical approximation of the volume integral is no longer broken up into multiple passes, as it was in previous approaches. Although this certainly is an improvement, due to the less tight control, the present framework lacks the capabilities to focus program flow on the relevant data. In fact, due to the GPU-native SIMD programming model, the slowest ray will now determine the rendering speed. Our paper seeks to provide solutions to enforce more control over the rendering process, with the special SIMD programming model in mind, while retaining this new and promising single-pass rendering paradigm. Our efforts were rewarded with speedups of up to 8.

1. Introduction

Volume rendering has become an integral technology in many areas of science, engineering, and medicine, and even entertainment thanks to the emergence of programmable graphics hardware (GPUs), available at affordable cost everywhere. The use of graphics hardware for volume rendering began with SGI workstations equipped with 2D and 3D texture mapping hardware and progressed to early generations of PC-grade graphics boards [RSKB*], which properly shifted and composited a set of axis aligned slices in front-to-back order. Pre-integrated volume rendering [EKE01] was later introduced to cope with under-sampling artifacts, that were produced when the above scheme was used with oblique viewing directions. The accelerated viewing capabilities enabled interactive transfer function design with a multi-dimensional framework [KKH02]. Acceleration techniques, such as early ray termination and empty space skipping [LMK03], and hierarchical acceleration structures [GWGS02] were also introduced. The slice-based rendering scheme was retained until the render-to-texture capabilities emerged, facilitating the storage of intermediate results onto textures and effectively driving program flow. This, in fact, gave rise to the ubiquitous GPGPU movement, and enabled the departure from the slice-compositing scheme of the early years. A noteworthy contribution in this regard is the paper by Krueger and Westermann [KW03], who implemented a ray caster on the GPU. This effort was followed by Weiskopf et al. [WSE04], who extended this framework to non-linear ray tracing. Both methods explicitly enforced the program flow by rendering control polygons for every major step of the ray casting algorithm, using textures to hold the intermediate computation results. The repeated sequence of individual steps are: advancing the rays, interpolation of samples in the 3D data texture, shading, compositing. While the programming model of graphics hardware is already SIMD (Single Instruction/Multiple Data, where the parallel processor performs the same operations on different data), the explicit decomposition and enforcement of the volume rendering process into these various pipeline steps created an even stronger SIMD lockstep processing mode. This gave rise to significant overheads associated with rendering the control polygon at each step, and requiring multiple passes. The upside to this is that due to the strong decomposition, rays that have become opaque could be eliminated (terminated) between steps and empty space could be culled [KW03]. Hadwiger et al.[HSS*05] presented a GPU-ray casting system for isosurface rendering which uses a block-based space-skipping acceleration scheme and performs screen-space shading. Splatting was also accelerated on the GPU using early-z culling [NM05]. Finally, Xue et al. [XZC05] introduced isosurface-aided hardware acceleration techniques to slice-based rendering, and Weiler et al. [WMKE04] provided a hardware accelerated ray-casting framework for rendering texture encoded tetrahedral strips.

The addition of loop and branch capabilities into the GPU programming set has enabled the more natural and free-flowing pipeline execution mode used in single-pass ray-casting [SSKE05]. In the new ray casting programs, a ray is first initialized and then steps across the volume and repeatedly executes all pipeline steps to integrate the discrete volume rendering integral. This only requires a single control polygon to be rendered, sending the ray fragments on their ways. We have found that the present framework lacks the capabilities to focus program flow on the relevant data, and, due to the GPU-native SIMD programming model, the slowest ray determines the rendering speed. We explore solutions to enforce better control over the rendering process, with the special SIMD programming model in mind, giving speedups of up to 8.

2. Single-Pass Ray-Casting

The latest generation of NVidia and ATI hardware, which support the DirectX Pixel Shader 3.0 API and NVIDIA's NV_fragment_program2 extension, enabled the introduction of single-pass ray-casting. This improved approach, thoroughly described in [SSKE05], takes advantage of the new branching and looping capabilities of the GPU in order to completely eliminate the need for intermediate passes required in previous implementations [KW03]. These additional passes were necessary to trace through the entire 3D volume, but also to stop rays that are outside of the volume or in already opaque regions.

Single-pass ray-casting is significantly faster than multi-pass ray-casting when rendering in semi-transparent mode, where very few rays will terminate due to opacity. Since theoretically the two approaches have exactly the same complexity, the difference in performance is defined by their overheads. In this case, the single-pass approach has much less overheads.

In certain scenarios, such as iso-surface rendering, and in cases where the opacity accumulates faster, depending on the transfer function, the multi-pass approach can be expected to perform better, since certain hard-wired optimizations including early-z culling can be applied to eliminate fragments from rendering on subsequent passes. These optimizations take advantage of the local coherence of neighboring rays since the whole process is inherently synchronized to lock-step.

The current single-pass implementations, including the one by [SSKE05] rely on a similar property of the SIMD processing pipeline that performs the fragment processing. More specifically, all the fragments that are "in flight" (that is the group of fragments that are being processed at the fragment pipeline at a given moment), are essentially run in lock-step. This means that all the fragments included in this currently processing group share the same instruction counter. So, if these fragments are executing a data dependent loop, then all the fragments within this group will either be working, or in the worst case, idle-processing until all of the fragments are ready to exit the loop.

The exact size of the "in-flight" fragment region is undisclosed by hardware manufacturers as it varies across different graphics hardware models, brands, and different generations of the same model, but it is expected to be a few hundred in the GeForce 6 series GPU [KF05]. This, depending on the fragment positioning in certain regions of the volume, translates to a few pixels/rays being able to dramatically slow-down the performance of the whole scene. In fact, it was also observed in [SSKE05] that early - ray termination even when rendering iso-surfaces using a full volume shader does not give performance that is consistent with what we have observed with previous approaches that use multi-pass rendering combined with early-z culling.

To verify this assumption, we have tested the single-pass ray-casting approach on a full volume rendering task but setting the transfer function to define iso-surface rendering.

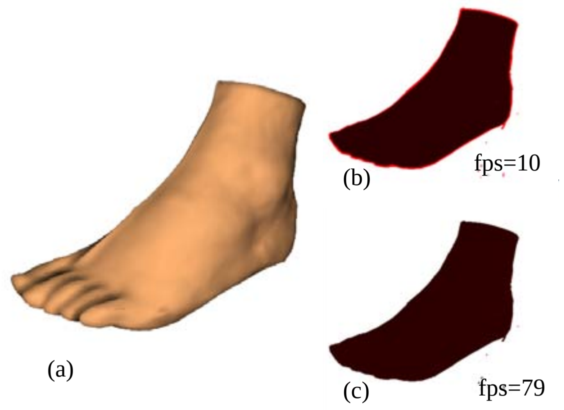

Figure 1: Example of SIMD ray-casting slowdown. (a) An iso-surface rendering of the foot dataset (viewport 512x512). Unoptimized frame-rate: 9.5 fps, Optimized frame-rate: 57fps. (b) Number of iterations per fragment for unoptimized SIMD ray casting. Darker red color equals to less iterations. (c) Number of iterations per fragment for optimized SIMD ray casting. We can notice from (b) that just a few fragments are causing a slowdown of about 8.

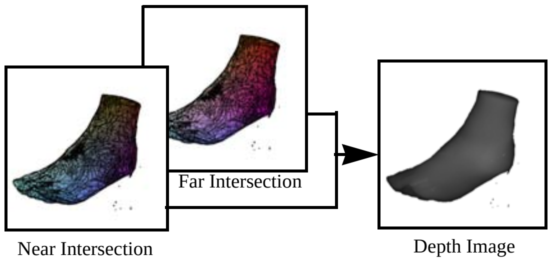

Figure 2: The data structures used in the PARC algorithm for SIMD raycasting. Here, the volume's outer surface mesh has been used to produce the near image texture and the far texture which store the intersection coordinates in RGB. The depth image helps to isolate the visible volume and avoid raycasting empty space.

Therefore, the alpha channel should saturate very quickly, and by adding a simple check inside the rendering loop should terminate the ray as soon as the alpha passes the opacity threshold. In the example shown in Fig. 1, we illustrate the number of iterations executed in each ray. In subfigures (b) and (c) darker shades of red depict less iterations for the pixel. We can clearly see that a group of edge pixels, which do not accumulate enough alpha to reach the threshold, keep executing until they reach the volume bounds. Unfortunately, because of the way the fragments are processed inside the GPUs SIMD pipeline, these "rogue pixels" were able to slow down the whole scene down to about 8 times compared to the frame-rates of our suggested optimized implementation. The iteration counts of the optimized renderer are shown in Fig. 1c, where we can see that for the iso-surface case all the pixels in the scene render with similar very shallow rays.

Experiences such as the above clearly suggest the necessity for an optimization scheme which takes into account the specific characteristics of each volume dataset and ensures that the rendering pipeline is fully utilized most of the time.

3. Implementation

A very simple yet quite effective and efficient acceleration approach for ray casting was proposed as early as 1995 in [SA95]. The Polygon Assisted Ray Casting algorithm (PARC) uses the hardware pipeline to acquire the pixel locations of the front and back surfaces of the dataset. This is done by rendering a polygonal model of the volume into two different target textures, using a two-pass approach similar to depth-peeling, where depth-buffer testing is used to render the front-most and back-most surfaces separately. The resulting textures can then be used to define the beginning and end of each ray. In the original implementation of PARC, this information was later read in back to the CPU and used later to accelerate the software based ray casting engine.

Our suggested single-pass ray casting method uses a polygonal model that is computed only once, using a simple iso-surfacing approach such as the marching cubes [LC87], based on the opacity assignments of the current transfer function. The algorithm then introduces a setup step at the beginning of the pipeline, which renders the polygonal model and stores the entry and exit points for each ray of the scene, as the PARC approach requires. The size of the polygonal model is kept at about 100-150K triangles using the mesh simplification facilities provided by DirectX. When the volume dataset is larger than 128^3 we sub-sample the volume and then polygonize. The resulting surface is then slightly resized in order to include boundary features that might have been lost in the sub-sampling step. This conservative approximation of the outer surface ensures the inclusion of all features above the iso-surface threshold, while keeping the polygonal model to a manageable size.

The resulting front-location and back-location textures store the entry and exit positions for each ray in perspective viewing, and can then be used inside the single pass fragment shaders to lookup these values (See Fig. 2). This approach provides a very tight bound, which encloses all possible volume locations to be sampled and restricts computation only to these necessary locations. In addition, as shown in the illustrated example of Section 2, this also ensures that any “rogue rays” that don’t adhere to the terminating conditions do not keep processing until the volume’s bounding box is reached. In fact, the proper depths of the rays passing through the boundary voxels that were causing the original approach to stall are very small. Therefore these rays should not stay too long in the pipeline anyway.

The next step is equivalent to empty-space skipping. After the front and back location textures are computed, a depth setting pass is run in order to prepare the depth-buffer for early-z culling that will eliminate all rays with zero depths.

Following is the final step, which defines the ray casting pass. Inside the ray casting loop, each ray first looks up it’s starting and ending position using the front and back position textures, and sets up the termination conditions for the rendering loop. Since we are using DirectX Shader Model 3.0, the instruction counter of the loop can only go up to 255, so depending on the size of the volume and the step size of the ray, a second loop inside the main loop is used to adjust the final loop counters.

A flexible shader loading mechanism is available on the system, which allows the user to dynamically load new shaders to be used in the ray casting step of the pipeline. Current examples include a full volume rendering shader which is able to load the xml transfer function sets shared by our collaborating visualization projects, as well as X-Ray and MIP projection shaders and step counters for illustration purposes.

The front and back surface position textures are available to the volume shaders at all times. This would be useful in scenarios where the ray direction changes, such as refraction, where the exit point is different from what was computed in the beginning. In this case, the front and back position textures may be used by the shader at each step to re-compute direction and exit points inside the main rendering loop.

The whole system has been implemented using the DirectX framework, which provides some standardized functionality in handling mesh data-structures, 3D textures 2D textures that are easily attached to rendering surfaces and available depth-buffers, in a very similar fashion to frame buffer objects. Our implementation has also made it very easy to also handle non-uniformly sampled volumes by using simple adjustments. This is possible by just introducing an additional scaling transform onto the surface mesh before it is rasterized in the pre-processing pass. This step assures that all rays now conform to the loaded volume sampling rate along X, Y, Z directions and a minor adjustment is also done inside the shader when volume samples are fetched.

4. Performance Evaluation

We have evaluated our system implementation on a workstation equipped with a Athlon64 3200 2Ghz processor, with 1GB memory and the NVidia 7600GT graphics board. Our SIMD-aware ray caster was build using DirectX 9.0 and the fragment shaders were compiled using Pixel Shader V3.0.

Our proposed optimization approach uses the PARC algorithm which requires two rasterization passes of the surface mesh, in order to create the near and far intersections for all the rays. We have evaluated the overall overheads of our approach by rendering only the first two passes without the ray-casting step. The incurred costs averaged about 400fps, or about 2.5 mSec per frame for all the tested datasets. This is the only cost that our SIMD-aware approach imposes to the pipeline. In a full shaded volume rendering scenario, at 32-bit resolution which is now achieved by current single-pass ray-casting implementations, this cost is considered negligible, given the average rates of current approaches for a 512x512 viewport and volumes of up to 256^3. The average size of the mesh is about 100-150K triangles. If this mesh becomes larger than 150K triangles, then the system uses subsampling as described in Section 3 and the mesh simplification utilities provided within the DirectX suite to reduce the size of the mesh, in order to keep the overall overhead of our optimization approach below 4 mSec.

In the next part of our evaluation process we measure the overall impact of the optimization approach to performance. For the following measurements, we have used full-shaded volume rendering with half-step ray sampling and compared the single-pass ray-casting algorithm in 3 modes: (I) Naive algorithm, using volume bounding box and opacity culling, (II) Using volume surface with early-z culling for empty-space skipping, (III) SIMD-aware, using ray-entry and ray-exit points to restrict the number of ray iterations. All volumes were scaled to fully cover a 512x512 viewport window. The corresponding images are shown in Fig. 3 and frame-rates are tabulated in Table 1. The I,II,III columns show the frame-rates for the respective rendering modes, and the last two columns show (II*) the speedup of II over I, which is due to empty-space skipping, and (III*) the speedup of III over II, which is due to bounding the ray iterations.

In addition to the expected speedups resulting from empty space skipping (which is achieved by culling all fragments with no ray-entry point), the results in Table 1 show significant speedups ranging from 2 up to 8 times from bounding the number ray iterations. We can also notice that the datasets rendered in iso-surface mode gained higher speedups using this optimization. The wide range of speedups is easily explained by the fact that in transparent rendering most of the rays have to traverse the entire volume, thus the gain of using the ray length bounds provided by the algorithm has less impact.

Looking at the bottom half of the table, where the last four volumes are rendered in both transparent and iso-surface mode, we can compare the impact of opacity culling, which is present in all I,II,III rendering modes. It is clear

TABLE 1. Impact of SIMD-aware ray-casting approach. (I) Naive raycasting using volume bounding box (II) Empty-space skipping (III) SIMD-Aware with ray bounds (II*) Speedup of II over I, (III*) Speedup of III over II.

Volume	**Size**	**I**	**II**	**III**	**II***	**III***
Lobster	320^2x36	4.0	7.0	32.0	1.8	4.6
Bonsai	128^3	3.4	8.5	16.9	2.5	2.0
Aneurism	128^3	3.4	10	32.0	2.9	3.2
Foot Trs.	128^3	3.1	7.8	35.0	2.5	4.5
Foot Iso	128^3	3.6	10	79.0	2.7	8.0
Teddy	128^3	4.0	4.7	13.0	1.2	2.8
CT-Head	128^3	4.6	5.1	14.0	1.1	2.7
Engine trs	256^3	3.1	3.3	13.0	1.1	3.9
Engine iso	256^3	5.0	10.5	38.0	2.1	3.6
VisMale trs	256^3	3.9	4.5	10.3	1.2	2.3
Vis Male iso	256^3	3.2	5.5	30.6	1.7	5.6
Foot Cat. trs	256^3	3.2	3.7	11.0	1.2	3.0
Foot Cat. iso	256^3	5.4	5.5	17.0	1.0	3.1
Frog trs	256^2x44	3.2	6.9	25.0	2.2	3.6
Frog iso	256^2x44	3.3	11.3	37.0	3.4	3.3

from this part of the table, that opacity culling (early ray termination based on alpha accumulation) has much less impact on performance in rendering modes I and II, a result that is also consistent with what was observed in [SSKE05]. On the other hand, the SIMD-aware approach enables opacity culling to have a more significant impact on performance, and it is consistent with the results of the past slice based approaches.

Overall, these results justify the use the PARC algorithm to accelerate single-pass ray-casting despite the incurred costs of at most 2-4 mSec per frame.

Conclusions

In this paper we have explored solutions that address some of the problems of current single-pass ray-casting algorithms, which utilize the latest features on GPU hardware such as loop flow control and branch capabilities. We have identified that the main flaw of the current SIMD ray-casting systems lies with the fact that very few unbounded rays are allowed to slow down the entire scene. We have proposed the use of the PARC (Polygon Assisted Ray Casting) algorithm, in order to ensure that all rays are bounded to the limits of the volume's outermost surface. We also provided a performance analysis which shows that for the minimal cost of about 2-4 mSec per frame, one can gain up to 8 times speedup on typical volumes. We are currently focusing on a load-balanced SIMD-aware ray casting system.

Acknowledgements

This research was supported, in part, by NIH grant 5R21EB004099-02 and NSF-CAREER grant ACI-0093157.

References

[CCF94] B. Cabral, N. Cam, J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware," *Proc. Symposium on Volume Visualization '94,* pp. 91.98, 1994.

[CN93] T. Cullip, U. Neumann, "Accelerating Volume Reconstruction With 3D Texture Hardware," *Tech. Rep. TR93-027,* Univ. of North Carolina at Chapel Hill, 1993.

[EKE01] K. Engel, M. Kraus, T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01,* pp. 9.16, 2001.

[GWGS02] S. Guthe, M. Wand, J. Gonser, W. Straßer, "Interactive Rendering of Large Volume Data Sets," *Proc. of IEEE Visualization '02,* pp. 53-60, 2002.

[HSS*05] M. Hadwiger, C. Sigg, H. Scharsach, K. Buehler, M. Gross, "Real- Time Ray-Casting and Advanced Shading of Discrete Isosurfaces," *Proc. Eurographics '05,* pp. 303.312, 2005.

[KF05] E. Kilgariff and R. Fernando. "The GeForce 6 Series GPU Architecture", in *GPU Gems 2, Addison-Wesley* 2005.

[KKH02] J. Kniss, G. Kindlmann, C. Hansen, "Multidimensional Transfer Functions for Interactive Volume Rendering," *IEEE Trans. on Visualization and Computer Graphics,* vol. 8, no. 3, 270-285, 2002.

[KW03] J. Krueger, R. Westermann, "Acceleration Techniques for GPUbased Volume Rendering," *Proc. of IEEE Visualization '03,* pp. 287.292, 2003.

[LC87] Lorensen, W.E. and Cline, H.E., "Marching Cubes: a high resolution 3D surface reconstruction algorithm", *Comp. Graph., 21-4,* pp 163-169 (SIGGRAPH), 1987.

[LMK03] W. Li, K. Mueller, A. Kaufman, "Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering," *Proc IEEE Visualization '03,* pp. 317-324, 2003.

[NM05] N. Neophytou and K. Mueller, "GPU accelerated image aligned splatting," *Volume Graphics Workshop 2005*, pp. 197-205, June 2005.

[RSKB*] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-textures and Multi-stage Rasterization," *Proc. ACM SIGGRAPH/EUROGRAPHICS Work. on Graph. Hardware,* pp. 109-118, 2000.

[SA95] L. Sobierajski and R. Avila, "A Hardware Acceleration Method for Volumetric Ray Tracing," *Proc. Visualization`95*, pp. 27-34, 1995.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, T. Ertl, "A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting," *Volume Graphics Workshop '05,* pp. 187-195, 2005.

[WMKE04] M. Weiler, P. N. Mallón, M. Kraus, T. Ertl, "Texture-Encoded Tetrahedral Strips", *Symposium on Volume Visualization 2004,* pp. 71-78.

[WSE04] D. Weiskopf, T. Schafhitzel, T. Ertl, "GPU-Based Nonlinear Ray Tracing," *Comput. Graph. Forum,* vol. 23, no. 3, pp. 625-634, 2004.

[XZC05] D. Xue, C. Zhang, R. Crawfis, "Isbvr: isosurface-aided hardware acceleration techniques for slice-based volume rendering." *Volume Graphics 2005,* pp. 207-214.