GPU Accelerated Voxel Traversal using the Prediction Buffer

Colin Braley *
Virginia Tech

Robert Hagan *
Virginia Tech

Yong Cao *
Virginia Tech

Denis Gracanin *
Virginia Tech
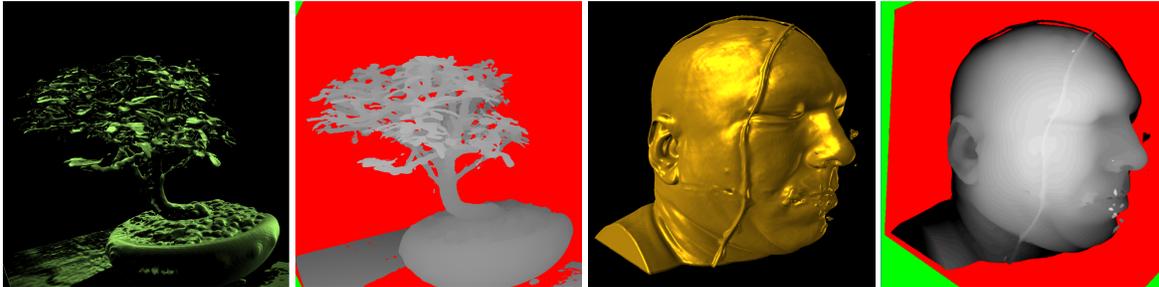
Figure 1: *The rendered iso-surfaces and the corresponding prediction buffers for two datasets: Bonsai and Visible Human® male.*

Abstract

The ever increasing size of data sets for scientific and medical visualization demands new isosurface volume rendering techniques to provide interactivity for the large datasets. The main obstacle to achieving interactivity is the computational bottleneck due to the dataset traversal and the corresponding amount of data transfer. We propose a novel GPU based dataset traversal technique that uses a prediction buffer to reduce the traversal time during dataset rotation. The reduction in the traversal time improves interactivity and consequently provides better insight into the dataset characteristics. We use a highly parallelized ray-casting algorithm and the proposed traversal technique to double the rendering speed. The factors which influence the rendering speed include block size, shared memory usage, and texture versus global memory. These factors were carefully considered to efficiently map the ray-casting volume rendering algorithm and the traversal technique to the GPU providing a high performance implementation.

CR Categories: I.3.1 [Computing Methodologies]: Computer Graphics—Hardware ArchitectureI.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism

Keywords: GPGPU, Isosurface, Volume Render, Depth Buffer, Rotation, Coherence, Frame-To-Frame Coherence , Variance, Prediction Buffer

1 Introduction

Volume rendering applications must be interactive for a user to effectively navigate and study a three-dimensional dataset. However, rendering the dataset in real-time presents a challenge since the datasets can be very large and complex and do not fit in the available texture memory. Consequently, the memory bandwidth limits the data transfer speed and reduces interactivity. We need novel methods to speed up the rendering process, improve interactivity and thus aid the user's ability to gain insight from analyzing the dataset.

Multi-resolution and downsampling based approaches can be used to address memory limitations. However, the resulting rendering, even at the highest resolution, has limited accuracy and smoothness.

Another way to address this problem is to look at the problem from the user perspective and identify typical ways the user interact with the dataset. The basic operations such as rotation, translation and scaling constitute majority of manipulations applied to the dataset and the corresponding visualization. Exploring characteristics of these basic operations may provide a way to address the problem.

Rotating a three-dimensional dataset is one of the main interactions used by the user to investigate all dataset segments. In a typical investigation, the user periodically rotates the dataset and then inspect various view angles until the dataset is fully examined. The lack of a response or a noticeable choppiness during rotation is likely to frustrate the user.

We propose a novel dataset traversal technique that enhances interactivity during the rotation stage of the viewing process. Due to the nature of rotation, the consecutive frames tend to be very similar. A prediction buffer storing the depth of each pixel in the previous frame is used together with the two new types of voxel traversals, sparse traversal and local traversal, to construct the next frame. However, the accuracy depends on the selected step size used during the traversal. The step size should be adaptive and based on the localized characteristics oh the dataset.

A novel variance bricking approach determines the variance of the dataset within a single brick. The brick variance value provides an estimate for the step size used during the dataset traversal for the points in that brick.

Although the technique introduces a slight decrease in image correctness during rotation, this decrease is negligible, especially given the following benefits of the technique:

- Enhances interactivity while rotating a datasets with novel predictive acceleration techniques.

- Effectively utilizes NVIDIA's CUDA memory hierarchy on the GPU.

- Utilizes variance preprocessing for adaptive step size selection.

2 Background

The two predominant methods for displaying isosurfaces are marching cubes, first described in [Lorensen and Cline 1987], and

---

*{ cbraley , rdhagan , yongcao , gracanin } @vt.edu

isosurface ray-casting, first described in [Levoy 1988]. Our research will focus on ray-casting, a type of *direct volume rendering*, since it does not require the large overhead of an intermediate surface representation. Furthermore, our method concentrates on accelerating the execution of the ray-casting approach to improve interactivity of the visualization.

Due to the highly parallelizable nature of the ray-casting algorithm, the implementation is highly suitable to the massively parallel nature of NVIDIA's CUDA framework on the GPU. The following sections outline background in the CUDA memory hierarchy, dataset traversal, and coherence-based techniques.

2.1 CUDA Architecture

We provide a brief background knowledge of Graphics Processing Units (GPU) here, so that we will have a better understanding of the implementation issues of our approach.

GPU is initially developed for accelerating 3D graphics applications. In the recently years, the computational power of GPU has grown quickly by introducing a massive number of parallel processing cores. The state-of-art GPUs exhibit more than 10 times of FLOPS in computation than multi-core CPUs. The GPU vendors also transform the GPU from a specialized processor to a general-purpose graphics processing unit (GPGPU), such as the NVIDIA GTX 280, which is also the GPU used in this paper. GTX 280 contains 30 streaming multiprocessors, each with eight cores per multiprocessor, totaling 240 cores, as shown in Figure 2. This GPU comes with one GB of global memory and supports a theoretical memory bandwidth of $141.7\frac{GB}{sec}$.

To lessen the steep learning curve, NVIDIA also releases a programming environment, the Compute Unified Device Architecture (CUDA), which can manage tens of thousands of threads that are executed in parallel. In CUDA, threads are grouped into blocks in either a 1D, 2D, or 3D configuration. Blocks are grouped into a grid in either a 1D or 2D configuration. In CUDA, up to eight blocks can be assigned to each streaming multiprocessor, provided that the multiprocessor has enough resources, including registers, shared memory, etc. Furthermore, each block is subdivided into 32-thread units called *warps*. Warps are the fundamental unit of scheduling, and divergent branching within a warp carries a high performance penalty. All threads in a warp are executed in parallel, until a divergent branch is encountered. Once a divergent branch is encountered, each separate path is executed sequentially. This is where the high branching penalty mentioned in Section 4.6.
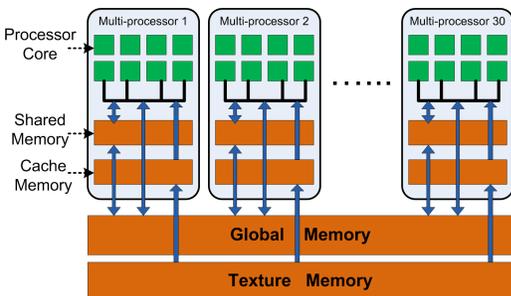

Figure 2: *The NVIDIA GTX280 GPU architecture.*

2.2 Ray-cast Based Volume Rendering

In the early stages of development of our GPU-based isosurface volume renderer, we benchmarked our application to evaluate runtime performance. Similar to other previous research, our analysis identified voxel grid traversal as the computational bottleneck. Other previous work has proposed various techniques for voxel traversal. The classic voxel traversal algorithm, presented in [Amanatides and Woo 1987], is very efficient in terms of using few floating point operations. However, this algorithm is branch intensive, and therefore it does not perform as well on modern GPU architectures, which have a heavy performance penalty for divergent branching.

Many techniques have been proposed to accelerate voxel traversal due to its large computational expense. Broadly, these techniques fall into two areas: *spatial subdivision* and *coherence* approaches. Spatial subdivision techniques perform space leaping using a special data structure. Space leaping avoids traversing areas of the volume data deemed unimportant. The definition of unimportant varies depending on what type of volume rendering is being performed. In isosurface volume rendering, unimportant areas are sections of the voxel data that do not include the specified iso-value. Space leaping based on min-max octrees was first proposed by [Wilhelms and Van Gelder 1992] with their Branch-On-Need Octree (BONO). This approach was later extended to kd-trees by [Wald et al. 2005].

Coherence-based approaches accelerate volume rendering by exploiting the strong temporal (frame-to-frame) or spatial (ray-to-ray) coherence found in volume rendering. We focus on temporal coherence due to its ability to accelerate navigation and rendering of animated datasets. [Yagel and Shi 1993] first used temporal coherence to speed up rendering with the C-buffer (coordinate buffer) technique. This C-buffer stores the coordinates of the nearest opaque voxel for use with semi-transparent rendering. After the viewpoint is moved, these C-buffer values are transformed in order to create the new starting point for voxel traversal. However, this can lead to propagation of frame-to-frame aliasing [Yoon et al. 1997]. Later, [Yoon et al. 1997] introduced the image cache data structure for optimizing rendering of isosurfaces. They utilize an iso-map as a modified depth buffer for acceleration of volume rendering. When the view is rotated, the iso-map is scanned along the axis of rotation in order to locate a new depth at which to begin traversal. While this approach is novel, and performs well on CPU architectures, it is not suited to GPU architectures due to its lack of data parallelism. More recently, [Klein et al. 2005] introduced a GPU-based approach to accelerate rendering using temporal (frame-to-frame) coherence. In this work, the authors stored a buffer of three-dimensional points representing the location of the first voxel along each ray that contributes to the rendered image. After some rotation or translation of the view has occurred, these points are then reprojected into the image plane and used as starting positions for voxel traversal.

3 Overview

In this work we are concerned with creating an isosurface volume renderer capable of rapidly producing high quality visualizations. Specifically, the user should be able to examine the data in real-time. Our volume renderer is based on a ray-cast approach, where a ray is cast at the volume data for each pixel in the final rendering image. The costliest computation step in ray-cast algorithms is voxel traversal. By focusing on this performance bottleneck in ray-cast based volume renderer, our primary contribution is the novel usage of a *prediction buffer*, a modified depth buffer, to accelerate voxel traversal.

For each ray cast, the prediction buffer stores the distance along the ray at which it intersects the isosurface. The prediction buffer also stores special values to indicate situations such as when a ray misses the bounding box containing the volume data, or when a ray intersects the bounding box but misses the isosurface.

During an interaction between the user and volume renderers, there are usually a small transformation changes between the consecutive frames. Among all transformation-based manipulations, we focus on rotation, which is the most commonly used transformation for user interaction. When the user rotates camera from point A to point B by a small angle, $d\theta$, only small changes in prediction buffer values usually occur. Utilizing this frame-to-frame coherence in the prediction buffer, we introduce two new versions of voxel traversals, *sparse traversal* and *local traversal*, in order to accelerate the rendering process. The idea is to use the prediction buffer calculated from the previous rendered frame to accelerate the process of voxel traversal.

In addition to presenting the prediction buffer, we also outline how to map our proposed iso-surface volume rendering algorithm, including prediction buffer calculation, onto current GPU architectures. We conduct an exhaustive performance analysis, and discuss which factors affect the performance of a volume rendering application built with NVIDIA's CUDA toolkit.

4 Prediction Buffer

In this section we first provide a description of our overall rendering algorithm with respect to prediction buffer. We then illustrate sparse traversal and local traversal algorithms in detail. In the end, we describe how to map the computation of our prediction-buffer algorithm onto NVIDIA's GPU architecture.

4.1 Prediction Buffer Algorithm Overview

In our volume rendering algorithm, we cast one ray for each pixel in the final rendered image. In a standard voxel traversal process, the voxels are tested one by one, starting from the first voxel hit by a ray until the last voxel intersected with the same ray. Our prediction buffer is designed to record the ray-cast results from the previous frame: 1) hit the iso-surface; 2) miss the iso-surface but hit the volume bounding box; 3) miss the volume bounding box, and; 4) no information (e.g. the first frame). For a specific pixel p_{ij} in the final image, we record its *depth* value, the distance d_{ij} from the *eye* to the interaction point along the ray, as its prediction buffer value. If the ray does hit the iso-surface (case #1), the prediction buffer value d_{ij} is a positive number. When the ray misses the isosurface but hits the bounding box containing the volume data (case #2), we store a negative flag value, $d_{hitBoxMissSurf}$ in p_{ij}. However, when $\vec{R}_{ij}$ misses the voxel bounding box completely (case #3), we store a different negative flag value, $d_{missBox}$. Lastly, a value in the prediction buffer can take on the value d_{noInfo}. This value indicates that the prediction buffer has no information about the pixel in question (case #4). This might arise when rendering immediately after the user has rotated the dataset by a very large amount, thus eliminating the ability to use coherence to speed up traversal. Additionally, operations such as changing the iso-value and changing the dataset invalidates the prediction buffer.

After the user rotate the camera for a small amount, $d\theta$, our prediction-buffer algorithm generates a color value for each pixel p_{ij}, and uses different traversal algorithms for the four different cases according to its prediction buffer value from last frame. In case #1, since the ray intersects with the iso-surface for the previous frame, we start voxel traversal from the ray distance indicated by the prediction buffer, assuming that the iso-surface is close to the intersection point from the last frame. The traversal algorithm that handles this case is called *Local Traversal*. In case #2, where we miss the iso-surface but hit the bounding box, it is reasonable to assume there is higher possibility that the ray will miss the iso-surface again. Therefore, we only sparsely step along the ray searching

for intersection with iso-surface. This traversal algorithm is called *Sparse Traversal*. In case #4, since we have no information about iso-surface intersection of the ray from last frame, we simply conduct a standard ray traversal, which is called *Full Traversal* in our algorithm. Of course, we don't need to handle case #3.

The overview of the our prediction-buffer algorithm is shown in Algorithm 1. We give the detailed description of *Sparse Traversal* and *Local Traversal* in Section 4.3 and Section 4.4, respectively.

Algorithm 1: High-Level Rendering Algorithm for rendering a pixel p_{ij}

Input : Two integers i, j specifying the pixel coordinates
Output: A color to shade the current pixel with

$d_{ij} \leftarrow predictionBufferGet(p_{ij})$
$\vec{R} \leftarrow createRay(p_{ij})$
$t_{intersect} \leftarrow d_{missBox}$

if $d_{ij} \geq 0.0$ **then**
    $t_{intersect} \leftarrow localTraversal(\vec{R}, t_{intersect})$
else if $d_{ij} = d_{hitBoxMissSurf}$ **then**
    $t_{intersect} \leftarrow sparseTraversal(\vec{R}, t_{intersect})$
else if $d_{ij} = d_{missBox}$ **then**
    $t_{intersect} \leftarrow d_{missBox}$
else if $d_{ij} = d_{noInfo}$ **then**
    $t_{intersect} \leftarrow fullTraversal(\vec{R}, t_{intersect})$
end

if $t_{intersect} \geq 0.0$ **then**
    $p_{ij} \leftarrow phongShade(getPointOnRay(\vec{R}, t_{intersect}))$
else
    $p_{ij} \leftarrow color_{background}$
end

$predictionBufferSet(p_{ij}, t_{intersect})$

Our algorithm utilizes $phongShade(\ldots)$, which requires calculation of a precise intersection point and surface normal. Since this is not the focus of our research, we present these details in Appendix A.

4.2 Full Traversal

Our method uses full traversal, the simplest traversal algorithm, when the prediction value from the prediction buffer d_{ij} has no information. In order to ensure the highest quality image when the camera is static, we also use full traversal when the dataset is not rotating. For some pixel p_{ij}, our method performs full traversal when the associated prediction buffer value is d_{noInfo}. This algorithm is a straightforward implementation from [Amanatides and Woo 1987], which presents a fast DDA-based 3D voxel traversal algorithm. This type of traversal ensures correct results because it tests every voxel along the ray to see if it includes an isosurface intersection.

4.3 The Sparse Traversal

Sparse traversal steps along the ray $\vec{R}$ with some predefined step size dt. At every step the iso-value of the voxel data is sampled at that point. If the iso-value is contained in the range between the current sample value and the previous sample value, the interval contains the iso-value. In this case, our method uses a few iterations of the bisection method to find the surface location accurately. We present psuedo-code for sparse traversal in Algorithm 2.

Algorithm 2: Sparse Traversal

Input : A ray $\vec{R}$, some floating point value dt, and an isovalue ρ_{iso}

Output: A distance along $\vec{R}$ to an isosurface intersection, or -1.0 if the surface was missed.

$t_{min} \leftarrow rayBoxIntersectFront(\vec{R}, voxelData)$
$t_{max} \leftarrow rayBoxIntersectBack(\vec{R}, voxelData)$

$t_{prev} \leftarrow t_{min}$
$t_{curr} \leftarrow t_{min} + dt$

$\rho_{prev} \leftarrow trilinearSample(t_{prev})$
$\rho_{curr} \leftarrow trilinearSample(t_{curr})$

while $t_{curr} \leq t_{max}$ **do**
    if $\rho_{iso} \in [\rho_{prev}, \rho_{curr}]$ **then**
        return $bisectionFind(t_{prev}, t_{curr}, \rho_{iso})$
    end
    $t_{prev} \leftarrow t_{curr}$
    $t_{curr} \leftarrow t_{curr} + dt$

    $\rho_{curr} \leftarrow trilinearSample(t_{curr})$
    $\rho_{prev} \leftarrow trilinearSample(t_{prev})$
end

return -1.0

While sparse traversal has a probability of introducing artifacts into the final image, we minimize these errors and maximize acceleration by using variance in the data. The sparse traversal has a chance of skipping an isosurface, especially if dt is large. However, a carefully chosen dt can minimize the chances of these misses. Section 4.5 outlines a procedure for choosing dt based on the variance in the data.
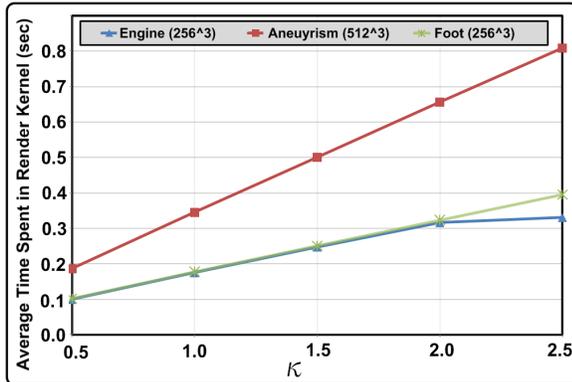

Figure 3: *The effect of κ on rendering time*

However, the most convincing argument for sparse traversal is that it is *much* faster than regular traversal (as presented in [Amanatides and Woo 1987]), on modern GPU architectures. This is true because of two factors. Firstly, branching can be a very computationally expensive operation on modern GPUs. In NVIDIA's CUDA architecture, divergent branching within a *warp* carries a high performance penalty. For a more detailed discussion of CUDA performance, see Section 2.1. The sparse traversal, obviously, contains no branching, which yields much higher performance. Even with branching out of the picture, some might argue that, particularly with a very small dt, this type of traversal will contain many calls

to an expensive $trilinearSample(\ldots)$ function. Regular traversal, on the other hand, guarantees that a minimum number of calls to $trilinearSample(\ldots)$ are made. Additionally, regular traversal would allow you to use a potentially less expensive bilinear sampling. However, on modern GPUs trilinear interpolation is implemented in hardware, so the performance penalty is negligible. Performance results for sparse traversal versus full traversal are presented in Figure 4.
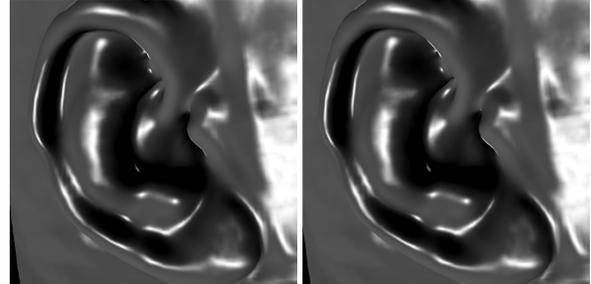
(a) Sparse Traversal $\kappa = 0.5$ (b) Full Traversal

Figure 4: *Visual Comparison - Sparse Traversal vs. Full Traversal*

Additionally, the loss in visual quality from sparse traversal is almost non-noticeable, provided κ is not too small. Figure 4, shows a dataset rendered fully with sparse traversal ($\kappa = 0.5$) and the same dataset rendered completely with full traversal. They are visually nearly indistinguishable.

4.4 The Local Traversal

The Local traversal is a variant of the sparse traversal and uses the frame-to-frame coherence found in the prediction buffer to make traversal faster. In this traversal, our traversal algorithm starts with the depth value indicated in prediction buffer, and performs sparse traversal by advancing on both side of the starting point. We traverse in this manner until we either locate the iso-surface, or both traversals (forward and back) exit the bounds of the volume data. We present the detail of *Local Traversal* in Algorithm 3.

However, it is possible that this type of traversal will produce different results compared with *Full Traversal*. For example, in Figure 5 we see a case in which this local traversal would not find the isosurface to the camera when rendering the red ray. However, we have found that these situations arise rarely in practice.
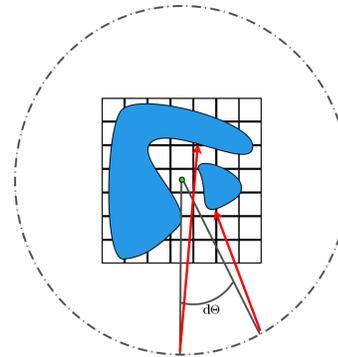

Figure 5: *Situation where local traversal will produce incorrect results*

In order to remedy this situation, our method performs full traversal every α frames. In practice, we found that an effective choice of α

depends on the dataset. In most of our experiences in this paper, we empirically choose $\alpha = 10$ by balancing between image quality and speed for the datasets we tested.

Algorithm 3: Local Traversal

Input : A ray $\vec{R}$, some floating point value dt, and an isovalue ρ_{iso}

Output: A distance along $\vec{R}$ to an isosurface intersection, or -1.0 if the surface was missed.

$t_{min} \leftarrow rayBoxIntersectFront(\vec{R}, voxelData)$
$t_{max} \leftarrow rayBoxIntersectBack(\vec{R}, voxelData)$

$t_{predict} \leftarrow predictionBufferGet(p_{ij})$
$t_{curr} \leftarrow t_{min} + dt$
$t_{prev} \leftarrow t_{min}$

$\rho_{curr} \leftarrow trilinearSample(t_{curr})$
$\rho_{prev} \leftarrow trilinearSample(t_{prev})$

while $t_{curr} \leq t_{max}$ **do**
    if $\rho_{iso} \in [\rho_{prev}, \rho_{curr}]$ **then**
        return $bisectionFind(t_{prev}, t_{curr}, \rho_{iso})$
    end
    $t_{prev} \leftarrow t_{curr}$
    $t_{curr} \leftarrow t_{curr} + dt$

    $\rho_{prev} \leftarrow trilinearSample(t_{prev})$
    $\rho_{curr} \leftarrow trilinearSample(t_{curr})$
end

return -1.0

4.5 Choosing dt through Variance Bricking

The stepping distance dt in *Sparse Traversal* and *Local Traversal* has a substantial impact on both the rendering frame-rate and image quality of our prediction-buffer algorithm. For a specific ray $\vec{R}_{ij}$, a smaller value of dt will make it more likely to find the accurate intersection point with the iso-surface. A larger value of dt will provide faster rendering time. We initially chose one global value of dt for the entire system. However, we found that a constant value was impractical, since the value depends on location of iso-surface with respect to each ray. If a ray passes through an area where the direction (measured by the normal vector) of the iso-surface changes frequently, a smaller value of dt is required. On the other hand, if there are less frequent changes in the direction of the iso-surfaces, a larger dt is enough to find the intersection point.

Calculating iso-surface normal for each surface point during run-time is a computationally expensive procedure. Instead, we relate the frequency of surface normal changes to the the variance v of voxel data with respect to other voxels in a brick.

$$v = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \qquad (1)$$

where n is the number of voxels in a brick and $\bar{x}$ is the mean of the data.

$$dt_{ijk} = \frac{\min(dx, dy, dz)}{\kappa_{ijk}}$$

where dx, dy, and dz are the length of a single voxel in the x, y, and z directions, respectively. The coefficient κ_{ijk} is determined by

function $f(v_{ijk})$, which describes the relation between the variance of a voxel and the dt value used during traversal through the voxel. In a preprocessing step on the CPU, we store the variance of the volume data for each brick. Areas with larger variance require a larger value of κ. However, our method also requires a function that maps from variance to κ values. We use the following function $f(x)$, which takes a variance v, and the maximum and minimum variances over all bricks, v_{max} and v_{min} respectively, as inputs.

$$f(v, v_{max}, v_{min}) = \kappa_{min}(v - v_{min})\frac{\kappa_{max} - \kappa_{min}}{v_{max} - v_{min}} \qquad (2)$$

Where κ_{max} and κ_{min} are experimentally determined constants. We found that values of $\kappa_{max} = 0.3$ and $\kappa_{min} = 0.1$ were good choices for reasonably sized data sets.

4.6 CUDA Implementation

We implemented a massively parallel isosurface volume renderer, featuring all the techniques described in this paper, using NVIDIA's CUDA framework, version 2.2. The majority of the code was written in C++, and the user interface was created with GLUT and GLUI.

In our system stores all volume datasets on disk in a binary format. The datasets are loaded at runtime and placed in floating point 3D textures. Our system consists of one main kernel function that performs isosurface volume rendering and writes the results to an OpenGL pixel buffer object.

Data Storage in GPU We test storage of the prediction buffer for several different methods. Initially, we store a single two-dimensional array of floats in global memory. At the beginning of each kernel call, a prediction would be read from global memory. At the end of each kernel call, a new prediction would be written. This implementation requires no blocking because two threads will never read the same prediction. We also implement the prediction buffer with two two-dimensional textures. Since threads inside of the same block are likely to access nearby pixels in the prediction buffer, there should be a large benefit from the spatial caching of the textures. However, since CUDA textures cannot be written to, this requires allocation of two identical sized pieces of global memory, A and B. At first, we bind A to a texture reference. Then, when we call our rendering kernel, we read from the texture which is bound to A and then write to B which is not bound to a texture. After the kernel call, we unbind the texture from A and bind it to B. On the next kernel call, we read from the texture, now bound to B and write to A. This process continues indefinitely, and is often referred to as double buffering. The double buffered texture implementation of the prediction buffer takes up twice the amount of memory as the global memory approach. We believe this extra memory use is tolerable because of the speed increase gained from textures. This tradeoff is acceptable because most of our memory is used to store the volume data. For instance, let us consider a volume rendering with an image plane of 1,024 by 1,024 and volume data with 512 by 512 by 512 samples. In this case, our volume data takes up $134,217,728$ bytes, while our prediction buffer takes up either $4,194,304$ bytes or $8,388,608$ bytes, depending on which storage approach is used. In the case where the texture approach is used, the prediction buffer takes up .05882% of the amount of memory the volume data takes up. In the case where the global memory approach is used, the prediction buffer takes up .03030% of the amount of memory the volume data takes up. As you can see, our memory usage is primarily concerned with storing the volume data itself.

Block Dimensions	Bucky (FPS)	Engine (FPS)	Porsche (FPS)
4 x 4	31	16	12
4 x 8	59	21	18
4 x 16	61	30	21
4 x 32	61	31	21
8 x 8	61	31	21
8 x 16	60	31	21
8 x 32	60	31	21
16 x 16	60	21	20

Table 1: *Performance results of three datasets for different thread block sizes.*

Computation to Core Mapping   Achieving high performance results with NVIDIA's CUDA relies largely on making intelligent choices for a few runtime variables. These variables comprise what is referred to as an *execution configuration*. An execution configuration consists of *block size*, *grid size*, and *shared memory size*, which can be determined when the computation in an algorithm is mapped onto GPU hardware. Our ray-casting maps a single pixel onto a single thread, and an optimal choice must be found for *block size* and *grid size*. Achieving high *occupancy* is one of goals of determining an execution configuration. Occupancy is defined on a per-multiprocessor basis as *number of warps being executed divided by number of warps allowed*. Maximizing occupancy results in higher utilization of the GPU. We utilized the CUDA occupancy calculator to determine suitable performance configurations, and we then carried out benchmarking to select the optimal configuration. Our results are shown in Table 1.

The Bucky dataset has dimensions 32 x 32 x 32, while the Engine and Porsche datasets are of sizes, 256 x 256 x 256 and 559 x 1024 x 347, respectively. Note that all FPS measurements have prediction buffer functionality disabled.

## 5   Performance Results and Discussion

We conduct the performance tests on a machine equipped with 2.33Hz Intel Core 2 Quad processor, 4GB of memory, and NVIDIA GTX280 GPU. In order to demonstrate the efficiency of our prediction buffer algorithm, we use five different datasets and compare the rendering performance of our proposed predict-buffer based traversal algorithm against the standard full traversal algorithm. The result is shown in Figure 6.

In order to perform a fair comparison, we implement both algorithms on GPU and use the exactly the same execution configuration (e.g. the same number of threads per block). During performance testing, we rotate the data volume by small angle, $d\theta = 1^o$, for every frame, and record the traversal time for each frame. We use the average traversal time for all rendered frames as our performance data.

6   Conclusion

In this paper we described a novel GPU-based traversal technique that, when used in conjunction with our *prediction buffer*, can greatly increase interactivity of iso-surface visualization for volume data. We also provided important implementation details for a high performance system using NVIDIA CUDA that can be used to create a high performance, GPU based, iso-surface raycasting system.

We used various datasets to test the developed technique. The results demonstrate its applicability to a wide variety of time-dependent or static datasets. Future work will focus on refining
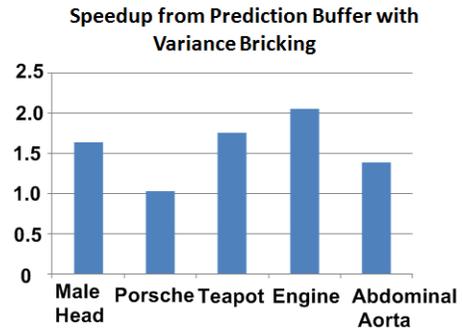
Figure 6: *Prediction Buffer Results*

the mapping function that determines the step size based on the variance value.

7   Acknowledgments

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *In Eurographics 87*, 3–10.

BAVOIL, L., AND SAINZ, M. 2009. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH '09: SIGGRAPH 2009: Talks*, ACM, New York, NY, USA, 1–1.

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, ACM, New York, NY, USA, 1–1.

CHEN, S. E., AND WILLIAMS, L. 1993. View interpolation for image synthesis. In *Proc. SIGGRAPH 93*, 1–6.

HADWIGER, M., SIGG, C., SCHARSACH, H., BHLER, K., AND GROSS, M. 2005. Real-time ray-casting and advanced shading of discrete isosurfaces. In *In Eurographics 05*, 303–312.

KLEIN, T., STRENGERT, M., STEGMAIER, S., AND ERTL, T. 2005. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Visualization, 2005. VIS 05. IEEE*, 223–230.

KONTKANEN, J., AND LAINE, S. 2005. Ambient occlusion fields. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 41–48.

LANGER, M., AND BLTHOFF, H. 2000. Depth discrimination from shading under diffuse lighting. *Perception 29*, 649–660.

LEVOY, M., 1988. Display of surfaces from volume data.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 163–169.

MARMITT, G., KLEER, A., WALD, I., FRIEDRICH, H., AND SLUSALLEK, P. 2004. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, 429–435.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 97–121.

PHONG, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18*, 6, 311–317.

REINBOTHE, C., BOUBEKEUR, T., AND ALEXA, M. 2009. Hybrid ambient occlusion. *EUROGRAPHICS 2009 Areas Papers*.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 75–82.

SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 73–80.

WALD, I., FRIEDRICH, H., MARMITT, G., AND SEIDEL, H.-P. 2005. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5, 562–572. Member-Slusallek, Philipp.

WILHELMS, J., AND VAN GELDER, A. 1992. Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3, 201–227.

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An efficient and robust ray-box intersection algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, 9.

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

YOON, I., DEMERS, J., KIM, T., AND NEUMANN, U. 1997. Accelerating volume visualization by exploiting temporal coherence. In *Proc. IEEE Visualization 97 LBHT*, IEEE Press, 21–24.

A Intersection and Surface Normal Calculation

Our algorithms perform Phong shading [Phong 1975], which requires a point in 3D space to calculate shading and an associated surface normal at that point. Our traversal algorithms only identify the voxel that contains the isosurface; they do not identify the precise sub-voxel depth of the surface. Since this is not the main focus of our work we will only touch on these topics briefly.

A.1 Ray-Voxel Intersection

In our system, we assume that the volume data varies linearly at the level of the individual voxel. Since the dataset sizes we target are large (greater than 256 voxels in each direction), this assumption does not result in poor image quality. Additionally, this assumption allows us to use the hardware based trilinear interpolation foud in graphics cards. More sophisticated methods for isosurface intersection are fond in [Marmitt et al. 2004].

The problem of ray-voxel intersection is stated as follows. Let some ray $\vec{R}$ pass through some voxel V. The ray $\vec{R}$ enters V at a parameter value of t_{in}, and exits V at a parameter value of t_{out}. The iso-values at these intersections are ρ_{in} and ρ_{out}, respectively. Now, assuming we are trying to render a surface of is-value ρ_{iso},

we know that if $\rho_{iso} \in [\rho_{in}, \rho_{out}]$ we can guarantee ρ_{iso} exists along $\vec{R}$ between t_{in} and t_{out}. Since we are assuming the value of ρ varies linearly at a sub-voxel level, this problem is simple.

In order to find t_{iso}, the t value at which the isosurface exists, we first use trilinear interpolation twice to find ρ_{in} and ρ_{out}. Note that this could actually be done with two bilinear interpolations, but trilinear interpolation is implemented in hardware so we use it instead. Now, to find t_{iso} we must solve a *single* linear equation. We know that as t varies from t_{in} to t_{out} along $\vec{R}$, the following equation holds:

$$\rho(t) = \frac{\rho_{out} - \rho_{in}}{t_{out} - t_{in}} t + \rho_{in}$$

We want to find the value of t at which this equation equals ρ_{iso}. Trivially, this is:

$$t_{iso} = \frac{\rho_{iso} - \rho_{in}}{\frac{\rho_{out} - \rho_{in}}{t_{out} - t_{in}}} + t_{in}$$

A.2 Surface Normal Calculation

In addition to finding the intersection point with the isosurface, we must also find the surface normal $\hat{n}$ at this point P. Recall from calculus that the gradient of some scalar field ρ, denoted by $\nabla \rho$, is a vector which points in the direction of *steepest ascent*. This vector expressed component-wise is simply:

$$\nabla \rho = \left\langle \frac{\partial \rho}{\partial x}, \frac{\partial \rho}{\partial y}, \frac{\partial \rho}{\partial z} \right\rangle$$

Since these are just first derivatives, we can approximate them with a *central difference*. In our system, we have implemented a simple $\mathcal{O}(h^2)$ accurate central difference scheme to calculate $\hat{n}$. This is expressed as:

$$\hat{n} \approx \quad \| \langle \frac{\rho(P_x + dx) - \rho(P_x - dx)}{2dx}$$
$$, \quad \frac{\rho(P_y + dy) - \rho(P_y - dy)}{2dy}$$
$$, \quad \frac{\rho(P_z + dz) - \rho(P_z - dz)}{2dz} \rangle \|$$

Note that dx, dy, and dz are the width of a single voxel in the x, y, and z directions respectively. Evaluating this central difference formula thus requires six calls to trilinear interpolation. Note that for edge cases where the sampling for the central difference requires non-existent data, we simply assume that the volume data's edge values repeat indefinitely (we do not extrapolate). This is made easier for us by CUDA's *texture addressing modes*. The addressing mode *cudaAddressModeClamp* performs this operation automatically.

We also implemented a higher order central difference that is $\mathcal{O}(h^4)$ accurate. These central differences are of the form:

$$f'(x) \approx \frac{-f(x + 2h) + 8f(x + h) - 8f(x - h) + f(x - 2h)}{12h}$$

We will not write this our in three dimensions simply for brevity's sake. This higher-order central difference will require 12 trilinear interpolations. However, in practice the overhead from these additional interpolations did not slow our renderer down significantly. For all of our test data, with dimensions ranging from 32x32x32 all

the way to 1024x768x256, the $\mathcal{O}(h^4)$ accurate central differences slowed our renderer down by between 0 and 2 frames per second.

Visually, we noticed very little differences in shading between the $\mathcal{O}(h^2)$ and $\mathcal{O}(h^4)$ central differences. Differences we only noticeable when zoomed in extremely closely, and when using the correct Phong shading parameters. These slight differences are illustrated in Figure 7. Overall, we don't think the higher order central differences are very important in practice because of the very slight performance hit, the increased programmer time required to implement them, and the marginal returns in terms of image quality.

Central Difference Comparison – Bucky Ball Data (32 x 32 x 32)

$\mathcal{O}(h^2)$ $\mathcal{O}(h^4)$
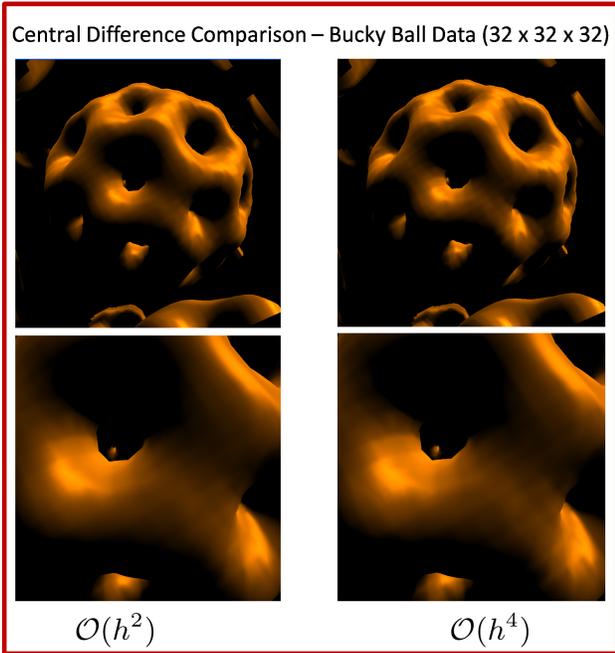
Figure 7: *Visual Comparison of Central Differences*

A.3 Examples: Rendered Images / Prediction Buffers
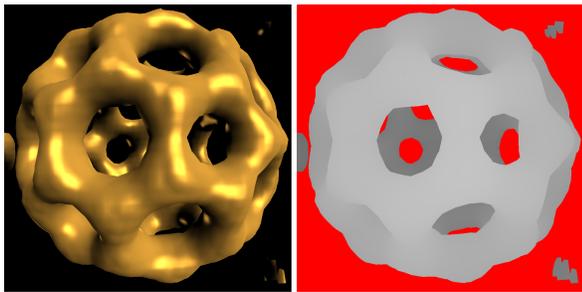

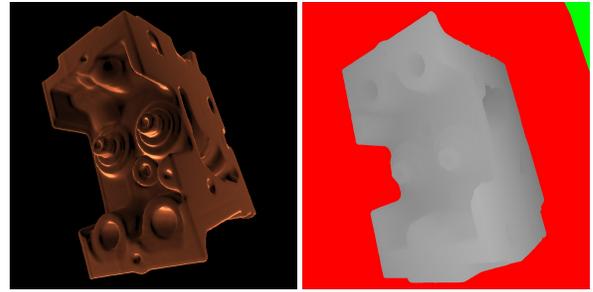**Figure 8:** *Bucky ball: the rendered image and prediction buffer.*

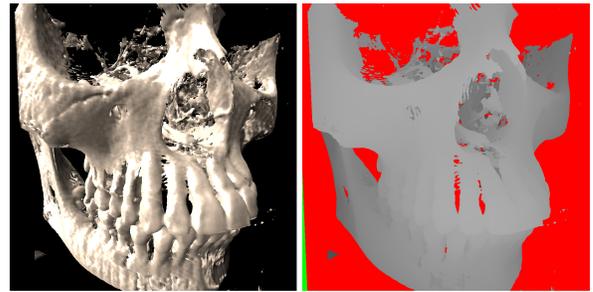**Figure 9:** *Engine: the rendered image and prediction buffer.*

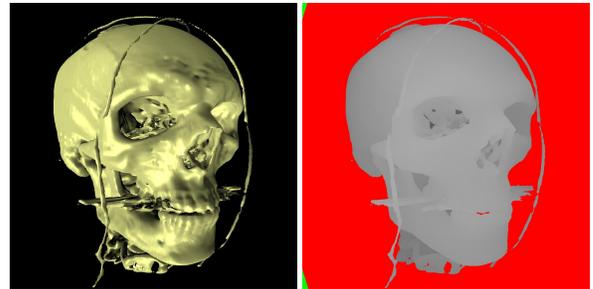**Figure 10:** *Skull 1: the rendered image and prediction buffer.*

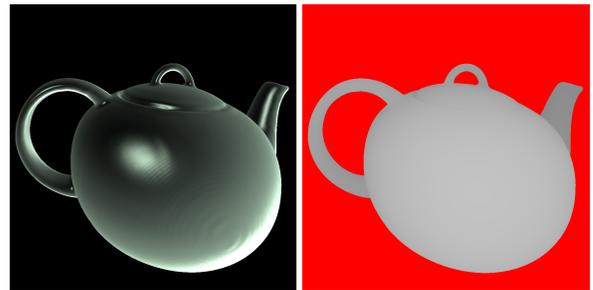

Figure 11: *Skull 2: the rendered image and prediction buffer.*

Figure 12: *Teapot: the rendered image and prediction buffer.*