GPU Accelerated Isosurface Volume Rendering Using Depth-Based Coherence

Colin Braley * Virginia Tech
Robert Hagan * Virginia Tech
Yong Cao * Virginia Tech
Denis Gračanin * Virginia Tech

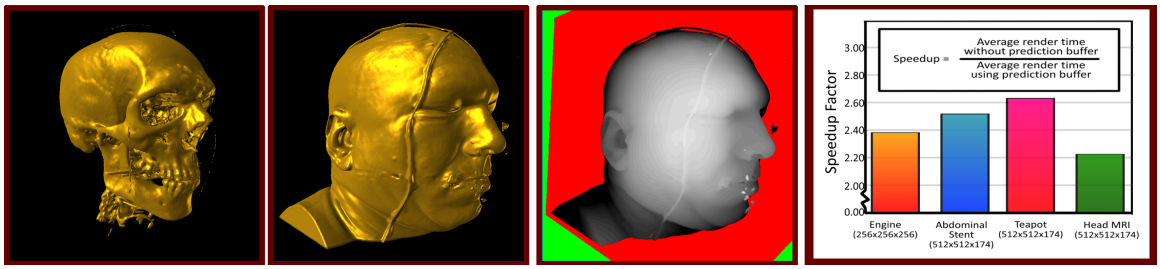

Figure 1: Two isosurfaces in the Visible Human® male dataset, a visualization of the *prediction buffer*, and a performance graph.

Keywords: GPGPU, Depth Buffer, Rotational Coherence

1 Introduction

With large scientific and medical datasets, visualization tools have trouble maintaining a high enough frame-rate to remain interactive. In this paper, we present a novel GPU based system that permits visualization of isosurfaces in large data sets in real time. In particular, we present a novel use of a depth buffer to speed up the operation of rotating around a volume data set. As the user rotates the viewpoint around the 3D volume data, there is much coherence between depth buffers from two sequential renderings. We utilize this coherence in our novel *prediction buffer* approach, and achieve a marked increase in speed during rotation. The authors of [Klein et al. 2005] used a depth buffer based approach, but they did not alter their traversal based on the prediction value. Our prediction buffer is a 2D array in which we store a single floating point value for each pixel. If a particular pixel p_{ij} has some positive depth value d_{ij}, this indicates that the ray R_{ij}, which was cast through p_{ij} on the previous render, intersected an isosurface at depth d_{ij}. The prediction buffer also handles three special cases. When the ray R_{ij} misses the isosurface, but hits the bounding box containing the volume data, we store a negative flag value, $d_{hitBoxMissSurf}$ in p_{ij}. When R_{ij} misses the bounding box, we store the value $d_{missBox}$. Lastly, when we have no prediction stored in the buffer, we store the value d_{noInfo}.

When rendering a specific pixel p_{ij}, we perform 1 of 2 different kinds of voxel traversals. If the prediction value is d_{noInfo}, we perform *full traversal*. If our prediction value is $d_{hitBoxMissSurf}$ we perform *sparse traversal*. For $d_{missBox}$ we do not traverse at all, and assume we have missed the isosurface. Lastly, when we some positive value for d_{ij} we perform *local traversal*. After traversal is complete, we will then update the prediction buffer.

2 Traversals

For full traversal we use the classic voxel traversal algorithm presented in [Amanatides and Woo 1987]. This algorithm is efficient in terms of floating point operations, but not in terms of divergent branching. Divergent branching is slow on the GPU seeing as the GPU is optimized for data-parallel operations. In NVIDIA Cuda, all intra-warp divergent branches are serialized, resulting in a large performance hit. This is our motivation for creating out alternate types of traversals. In sparse traversal, we step along the ray by intervals of dt. At every value step we sample the voxel data using trilinear interpolation done in hardware. If two consecutive samples bound the iso-value, we bisect this interval in order to locate the hit point. This process continues until we find the isosurface, or exit the voxel data's bounding box. This is significantly faster than full traversal since this greatly reduces the amount of branching, and because trilinear interpolation is very fast when implemented in hardware. The visual accuracy of this method relies on us choosing a small dt, while speed relies on choosing a large dt. We choose $dt = \frac{min(dx,dy,dz)}{\kappa}$, where dx, dy, and dz are the widths of a single voxel in the x, y, and z directions, respectively. κ is a user specified constant. Experimentally, we found $\kappa \approx 0.5$ to be a good trade-off. In local traversal, we have some positive predicted depth d_{ij} which our isosurface is likely to be near. In order to find the surface as fast as possible, we alternate back and forth, moving by some interval dt, starting at distance d_{ij}. Since it is likely that we will find an isosurface at a nearby location, rendering time is drastically reduced. However, this technique can create slight visual artifacts when an occluding isosurface appears. In order to remedy this, we perform a full traversal, for all pixels, every α renderings. Experimentally, we found $\alpha \approx 10$ to be a good trade-off.

3 Discussion of Results

In the above histogram, we see that our technique gets a relatively large speedup for many real-world sized data sets. This dimensionless speedup is simply the average time spent traversing the dataset when using full traversal, divided by the average time when using our prediction buffer based approach. We performed an automated 360 degree rotation around each dataset, and timed each Cuda kernel launch. We found that, while speedups were attained across the board, the speedup amount is slightly dependent on camera position and data characteristics. Note that in this data collection, and in all data in our accompanying video, $\kappa = 0.5$ and $\alpha = 10$.

References

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

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.

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