Parallel Methods for Isosurface Visualization

Charles D. Hansen, Tushar Udeshi, Steven Parker, and Peter Shirley
Department of Computer Science
University of Utah
50 S. Campus Center Drive Rm. 3190
Salt Lake City, Utah
84112-9205
{hansen | tudeshi | sparker | shirley}@cs.utah.edu

Extended Abstract

Many applications generate scalar fields $\rho(x, y, z)$ which can be viewed by displaying *isosurfaces* where $\rho(x, y, z) = \rho_{\text{iso}}$. Ideally, the value for ρ_{iso} is interactively controlled by the user. When the scalar field is stored as a structured set of point samples, the most common technique for generating a given isosurface is to create an explicit polygonal representation for the surface using a technique such as *Marching Cubes*. This surface is subsequently rendered with attached graphics hardware accelerators such as the SGI Infinite Reality. Marching Cubes can generate an extraordinary number of polygons, which take time to construct and to render. For very large (i.e., greater than several million polygons) surfaces the isosurface extraction and rendering times limit the interactivity. In this talk, we address this problem by two distinct methods. One method addresses the rendering problem by using multiple graphics adapters in parallel[1]. The second method generates images of isosurfaces directly with no intermediate surface representation through the use of ray tracing.

Hardware based Parallel Rendering

Current generation of high-end graphics adapters, such as the SGI Infinite Reality, have a quoted maximum polygon rendering rate of around 10M polygons/second. However, in practice this rate is difficult to achieve primarily due to geometry traversal (appropriate display list) and pixel coverage of the polygons. In exploratory scientific visualization, isosurfaces are typically represented by a large number of polygons. As the isovalue is changed, the resulting surface, and perhaps the topology, is modified making display lists impractical. As a result, when viewing isosurfaces of large data sets[2] rendering becomes a bottle neck. We have recently seen the attachment of multiple graphics adapters on high-end SGI Onyx2 supercomputers. The multiple graphics pipes can be used for multiple users or they can be combined to provide a single user the power of parallel rendering. Until recently, there has not been much experimentation with this configuration for polygon rendering.

At the University of Utah, we have implemented a technique for exploiting the parallel (up to eight) Infinite Reality graphics adapters. One can divide the image and assign each graphics adapter a subimage for rendering. This approach uses either preculling of visible polygons (software based frustum culling) or uses the hardware based visibility culling in the graphics adapters themselves. The problem with software based visibility culling is the dependency on spatial hierarchies for the underlying surface. When generating an isosurface, such pre-processing would be time-prohibitive. The difficulty with using the hardware based visibility culling is that each graphics adapter needs to process the entire set of polygons for its subimage. Using a *sort-last* rendering scheme provides a different approach. In *sort last* rendering, each graphics adapter renders only a portion of the polygons and the resulting partial images are combined, using depth comparison, to product the final result. This is the approach we have chosen to use since it provides better interactivity without the burden of preprocessing the isosurface once it is created.

Our algorithm proceeds as follows: each graphics adapter renders n/p polygons, where n is the total set of polygons representing an isosurface and p is the number of available graphics adapters. When this step is complete, the partial images need to be composited. We use the *binary-swap* method for image composition.

[1] The so called *SGI Onyx2 Reality Monster*.
[2] Large means greater than 1GBytes

This method composites in logarithmic time and utilizes all of the graphics adapters in parallel. Each graphics adapter swaps 1/2 of the remaining subimage and z-buffer with its partner, a stencil buffer is created where the new Z value is closer to the viewpoint than the old Z value and pixels are rendered from the new image using the stencil buffer which performs the correct hidden surface elimination. This process is repeated until all graphics adapters have exchanged subimages which results in a correct final image distributed among the different graphics adapters. The final stage collects the subimages to a single graphics adapter.

Surprisingly, the tradeoff for multi-pipe rendering occurs around 1M polygons. Given the large number of polygons extracted using typical isosurface methods, this clearly provides acceleration through the parallel rendering. The talk presents results for varying data sets and image sizes.

Ray-tracing Isosurfaces

Conventional wisdom holds that ray tracing is too slow to be competitive with hardware z-buffers. However, when rendering a surface from a sufficiently large data set, ray tracing should become competitive as its low time complexity overcomes its large time constant. The same arguments apply to the isosurfacing problem. Suppose we have an $n \times n \times n$ rectilinear volume which for a given isosurface value has $O(n^2)$ polygons generated using Marching Cubes. Given intelligent preprocessing, the rendering time will be $O(n^2)$. Since the performance improvement using multiple graphics engines only scales to a limited amount, namely the number of graphics engines, this seems a hard limit when using commercially available graphics accelerators unless a large fraction of the polygons are not visible. If a ray tracing algorithm is used to traverse the volume until a surface is reached, we would expect each ray to do $O(n)$ work. If the rays are traced on p processors, then we expect the runtime for an isosurface image to be $O(n/p)$, albeit with a very large time constant and a limit that p is significantly lower than the number of pixels. For sufficiently large n, ray tracing will be faster than a z-buffer algorithm for generating and rendering isosurfaces. The question is whether it can occur on an n that occurs in practice (e.g., $n = 500$ to $n = 1000$) with a p that exists on a real machine (e.g., $p = 8$ to $p = 128$). This talk demonstrates that with a few optimizations, ray tracing is *already* attractive for at least some isosurface applications.

Our algorithm has three phases: traversing a ray through cells which do not contain an isosurface, analytically computing the isosurface when intersecting a voxel containing the isosurface, shading the resulting intersection point. This process is repeated for each pixel on the screen. Since each ray is independent, parallelization is straightforward. An additional benefit is that adding incremental features to the rendering has only incremental cost. For example, additional shading techniques, such as shadows and specular reflection, can easily be incorporated for enhanced visual cues.

To find an intersection, the ray $\vec{a} + t\vec{b}$ traverses cells in the volume checking each cell to see if its data range bounds an isovalue. If it does, an analytic computation is performed to solve for the ray parameter t at the intersection with the isosurface:

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{\text{iso}} = 0.$$

When approximating ρ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in t. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell.

For the traversal of rays through the data, we found that traversing the cells is the computational bottleneck for large data sets, so we include optimizations to accelerate performance. The first optimization is to improve data cache locality by organizing the volume into "bricks" that are analogous to the use of image tiles in image-processing software and other volume rendering programs. The second is to use a multi-level spatial hierarchy to accelerate the traversal of empty cells. Cells are grouped divided into equal portions, and then a "macrocell" is created which contains the minimum and maximum data value for its children cells. This is a common variant of standard ray-grid techniques and the use of minimum/maximum caching has been shown to be useful. The ray-isosurface traversal algorithm examines the min and max at each macrocell before deciding whether to recursively examine a deeper level or to proceed to the next cell.

This work shows that ray tracing can be a practical alternative to explicit isosurface extraction for very large data sets. As data sets get larger, and as general purpose processing hardware becomes more powerful, we expect this to become a very attractive method for visualizing large scale scalar data both in terms of speed and rendering accuracy.