Parallel Multipipe Rendering for Very Large Isosurface Visualization

Tushar Udeshi and Charles D. Hansen*

Department of Computer Science
University of Utah
50 S. Campus Center Drive Rm. 3190
Salt Lake City, Utah
84112-9205
{tudeshi, hansen}@cs.utah.edu

Abstract. In exploratory scientific visualization, isosurfaces are typically created with an explicit polygonal representation for the surface using a technique such as *Marching Cubes*. For even moderate data sets, Marching Cubes can generate an extraordinary number of polygons, which take time to construct and to render. To address the rendering bottleneck, we have developed a multipipe strategy for parallel rendering using a combination of CPUs and parallel graphics adaptors. The multipipe system uses multiple graphics adapters in parallel, the so called *SGI Onyx2 Reality Monster*. In this paper, we discuss the issues of using the multiple pipes in a Sort-Last fashion which out performs a single graphics adaptor for a surprisingly low number of polygons.

1 Introduction

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*[8]. This surface is subsequently rendered with an attached graphics hardware accelerator, such as the SGI Infinite Reality. For even moderate data sets, 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. One approach to address this issue is to exploit parallelism.

The use of parallelism in computer graphics hardware is widely known. Most current generation graphics adaptors utilize parallelism in their design and implementation[1, 2, 9]. While these systems are extremely proficient at rendering geometry with superb rendering capabilities, such as texture mapping, the bottle neck for rendering large polygon sets is the speed at which polygons can be

* Contact Person for this paper

sent through the graphics pipeline[1]. Since there is a single thread which can send polygons to the graphics adaptor, the majority of rendering applications are serial and implicitly exploit the parallelism inherent in the graphics adaptor.

For scientific visualization of very large data sets, 512^3 and higher, parallel isosurface extraction and rendering techniques have been studied[3, 6, 7, 11]. These techniques exploit the large memory and parallelism of massively parallel computers to deal with the data explosion caused by scientific visualization of the large simulations running on the same machines. These techniques mimic, in software, the parallelism in the graphics hardware and achieve speedup although not at interactive rates. One of the problems with these approachs is the lack of an attached framebuffer or graphics adaptor for displaying and interacting with the image.

Clearly what is desired is a combination of these approachs which takes advantage of the large memory and parallelism provided by large-scale parallel computers and the interactive rendering capabilities provided by graphics hardware. Fortunately, we have recently seen the convergence of these two with the SGI ONYX2 which is an SGI Origin 2000 with attached InfiniteReality graphics adaptors[10]. SGI ONYX2 with multiple InfiniteReality graphics adaptors are called Reality Monsters. While these promise acceleration based on parallelism on both the macro scale (multiple graphic adaptors) and the micro scale (internal to each graphics adaptor), these systems are new and methods for exploiting them have not been studied. This paper addresses an approach to exploiting the multiple graphics adaptors for polygon rendering.

2 Parallel Graphics Hardware Approach

One approach to the classification of parallel rendering algorithms is to categorize based upon whether the parallelism is achieved in image-space or in object-space. However, many recent algorithms obtain performance by utilizing both image-space and object-space parallelism. A more useful taxonomy for parallel rendering which classifies rendering methods based on where data are sorted from object-space to image-space was presented by Molnar et al.[5]. A typical rendering process performs some *geometric* processing followed by some *rasterization* processing. Parallelization can take place during the geometric processing, during the rasterization processing, as well as pipelined parallelism between the two stages. At some point, primitives are sorted from object-space to image-space. Looking at where this sort takes place provides a useful method for classifying parallel rendering techniques. The sort to screen-space can take place before the geometric processing, after the geometric processing but before the rasterization, or after both the geometric processing and the rasterization. These methods are referred to as Sort-First, Sort-Middle, or Sort-Last.

The Sort-First approach exploits some *a priori* knowledge about which part of the screen the primitives will fall. This is utilized to send the primitives, pos-

[1] For 3D texture mapping based volume rendering, the bottleneck is typically pixel fill-rate. We limit our application to explicit polygon rendering.

sibly polygons, to the appropriate processor elements. Frame-to-frame coherence provides such knowledge. The screen is subdivided in some manner, typically interleaved, and each region is assigned a processing element which performs both geometry and rasterization without any need for communication. Sort-First suffers from load inbalance for both the geometry processing stage as well as the rasterization stage if primitives are not evenly distributed across the screen partitions.

The Sort-Middle approach needs no *a priori* knowledge and primitives, typically polygons, are distributed in some fair scheme between all the processing elements. Each processing element performs the geometric operations on its portion of the data. Following this stage, the transformed primitives are sent to the processor element responsible for the portion of the screen into which these primitives fall. Sort-Middle is typically well balanced during the geometry processing stage but suffers from load inbalance during the rasterization stage if primitives are not evenly distributed across the screen partitions. Additionally, there can be a communication bottleneck if all geometry processors are sending data to a single rasterization processor. The SGI Infinite Reality graphics adapter uses Sort-Middle parallelism internally in its graphics pipeline. Using current technology, Sort-Middle parallelism cannot be exploited with multiple pipes since each pipeline is totally independent from the rest.

Sort-Last rendering is sometimes referred to as an image compositing system. Primitives are distributed in some fair scheme among the processing elements. Each processing element performs both geometric processing as well as rasterization independent of all other processor elements. A local image is rendered on each processor element and the images are composited together to form a final image. In some systems, only active pixels from each subimage participate in the compositing phase. Sort-Last behaves particularly well with respect to load balancing since all primitives are fairly distributed at the beginning. However, the communication load for the image compositing phase can be quite severe and requires very high speed networks. In addition, transparency is non-trivial for Sort-Last systems. Still, the scalability for Sort-Last rendering makes it an attractive alternative for very large polygonal data sets such are those generated from large scientific data.

We have implemented a technique for exploiting the parallel (up to eight) Infinite Reality graphics adapters. One can use a Sort-First style approach by dividing the image and assigning 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. For exploratory scientific visualization where different isosurfaces are generated often and at interactive rates, such pre-processing for each resulting isosurface 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 multipipe Sort-Last rendering, each graphics adapter renders only a portion of the polygons and the resulting partial images are combined, using depth comparison, to produce 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.

The basic idea behind our algorithm is to divide the renderable data among the available graphics adapters, render each subset separately and *locally*, and combine the resulting partial images in an incremental fashion. This technique is strongly related to composition based volume renderers in that each graphics adapter renders a portion of the final image and these are combined[4]. It is similar to the composition network approach of Pixel Flow[9].

2.1 Binary Swap

Assume n is the total number of polygons representing an isosurface and p is the number of graphics adapters. Typically p is a power of two although this can be relaxed through simple extensions. We assume the isosurface, represented as n polygons, exists in shared memory. Each graphics adapter loads and locally renders n/p polygons. At this point, each graphics adapter has a partially complete image. These images are then composited onto the graphics adapter used for the final display. We use the *binary-swap*[4] method for image composition which composites the image in $log_2(p)$ steps. At each step, the graphics pipes send the top half of their active image to their partner and receive the bottom half from their partner. The partners are determined as being 2^i away where i is the composting step. Thus when utilizing 8 graphics adapters, for the first composite the partners are [(0,1), (2,3), (4,5), (6,7)] and for the second composite the partners are [(0,2), (1,3), (4,6), (5,7)].

Psudo-code of the algorithm follows. Assume p = number of pipes(numbered 0,1,...., p-1), q = the current pipe, H = height of image. For simplicity p is assumed to be a power of 2.

```
procedure binaryswap
begin
    let h = 0
    let levels = log2(p)
    let height = H
    for l = 0,1... (levels-1)
    begin
        let factor = 2^l
        let A be a band of the image from h to (h + height)
        if (l^th bit of q is 1) /* odd pipe */
            Send lower half of A to pipe (q - factor)
            Recieve and composite an image from pipe (q - factor) onto upper half of A
            let h = h + height/2
        else /* even pipe */
            Send upper half of A to pipe (q + factor)
            Recieve and composite an image from pipe (q + factor) onto lower half of A
        endif
        let height = height/2
        synchronize all pipes /* barrier */
    end
    send A to display pipe
end
```

Composition of the incoming image with the current image is done in hardware making use of the stencil buffer. This is achieved by the following sequence of steps:

1. Clear the stencil buffer to all zeros
2. Set the stencil operation to replace value to 1 if depth test passes and keep current value otherwise.
3. Draw depth buffer values. This will set the stencil buffer
4. Set stencil test to pass if stecil value = 1 and enable stencilling.
5. Draw the color buffer values.

This has the effect of maximizing hardware usage and scales better with image size than performing the composite with the CPUs. The active image is reduced by half at each step. At the end of $log_2(p)$ levels, the active image on each graphics adapter is composited into the display graphics adapter. Figure 1 shows the compositing steps from bottom to top. The bottom row shows the results of rendering the initial polygon distribution on each graphics pipe. The gray areas in the images are back facing polygons. The three compositing levels are the next three rows up. The compositing partners are shown with lines between the compositing levels.

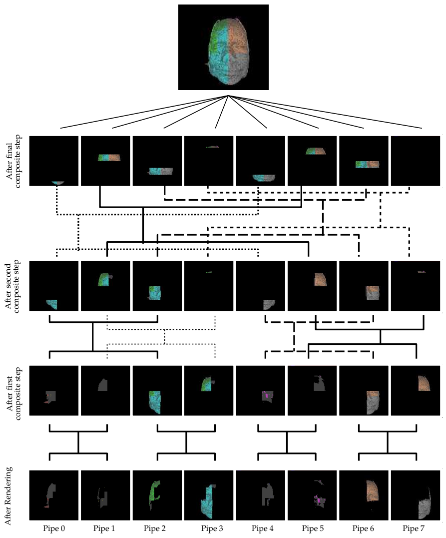

Fig. 1. The compositing levels are along the verticle axis and the graphics adapters are along the horizontal axis. After the final step (the top most level), the final image is composed from each of the partial images. (See color plate)

3 Experiments and Results

We have tested our technique with a variety of large polygon data sets. In this section, we present the results. In all tests, we compare our technique with the best (non-compositing) single pipe version. The rendering code is all written in OpenGL.

For the first example, we wanted to stress the multipipe rendering system to understand where the tradeoffs were for this technique. Recall, our goal is to render extremely large polygon data sets such as those one would see generated from 1GByte data sets. If we lower the number of polygons, we would expect the overhead from the multiple graphics adapters to limit the speed up. In fact, for sufficiently small polygon sets, we would expect the single graphics adapter to out-perform multiple graphics adapters. To test this, we take a 131,000 triangle isosurface generated from a CT scan. To test the scaling of polygon count, we instantiated multiple copies of this data, from 2 up to 8 copies. This provided a series of polygon data sets which varied from 131K to 1M polygons. We rendered these with 1 to 8 graphics adapters. To mitigate instantaneous timing anomalies, we rendered each frame 100 times. The results are shown in Figure 2 and Figure 3 for 512 x 512 and 1024 x 1024 images respectively. The X-axis is the size, in triangles, of the isosurface being rendered. The Y-axis is the time in seconds for rendering 100 images. As is shown in the plots, for the 1024x1024 image the single graphics pipe out performs the multple pipes for the isosurface containing 131,000 triangles. This is to be expected since the overhead of reading back the frame buffers and compositing in the multiple graphics adapters case adds overhead and the rendering speed of the Infinite Reality graphics adapters can easily handle that modest number of polygons. However, once the polygon count increases to 262K triangles, the multiple graphics adapters out perform the single graphics pipe for all cases. The improvement for multiple graphics pipes steadly increases as the polygon count increases. One can notice the difference between the 512x512 and 1024x1024 images. In the 512x512 case, the multiple graphics adapters are always faster than a single graphics adapter, even on this small number of polygons! The overhead of reading back and re-writing to the graphics adapters (the compositing step) for the larger images in the 1024x1024 case results in slower rendering times for low numbers of polygons. The overhead predominates for the 131K triangle case. However, as the polygon count increases to 1M triangles, the overhead becomes less of the overall time and the cost for rendering a 1024x1024 image reduces to near the cost of a 512x512 image.

For the next example, we extracted the isosurface representing the skin for the head portion of the visible woman data set (see Figure 1). The isosurface is composed of 1.4 million triangles. We again instantiated multiple copies of this data with 2, 3, and 4 copies resulting in 2.8M, 4.2M, and 5.6M polygon isosurface data sets. Figure 4 and Figure 5 show the results for both 512x512 and 1024x1024 images. Notice that the rendering times are very close with the 512x512 image being slightly faster for rendering with larger numbers of graphics pipes. This is due to the increased number of steps in the compositing operations with an increased number of graphics pipes. However, with even a modest 1.4M

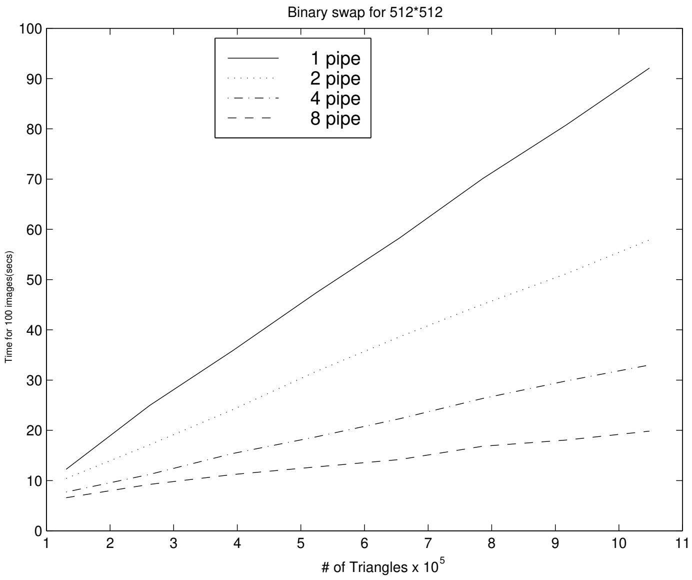

Fig. 2. Times for rendering a 512 x 512 image

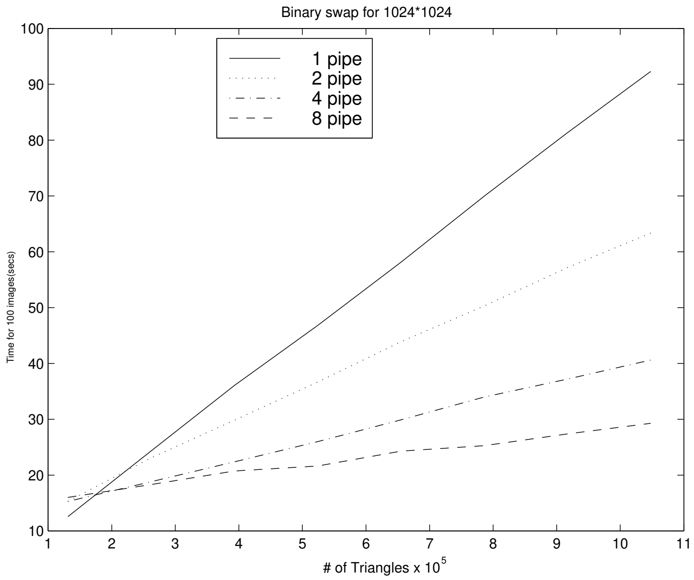

Fig. 3. Times for rendering a 1024 x 1024 image

polygons, 8 graphics adapters outperforms a single graphics adapter by a factor of 2. For 5.6M polygon data set, 8 graphics pipes outperforms, by 6 times, a single graphics pipe.

Tables 1 and 2 show the overhead caused due to the binary swap communication and compositing steps. For a small polygon count of 131,000, the overhead is fairly high. However for a polygon count of 5.6 M, these steps take less than 10% of the time thus resulting in huge improvements in time compared to the single pipe case.

Polygon count	131,000		5.6 Million	
Image size	512x512	1024x1024	512x512	1024x1024
Total time(secs)	10.4237	15.305	241.317	246.005
Overhead(secs)				
communication	1.248	5.640	1.248	5.640
composite	1.238	2.812	1.238	2.812
total overhead	2.486	8.452	2.486	8.452
% of total time	23.85%	55.2%	1.03%	3.04%

Table 1. Overhead for binary-swap and compositing compared to total rendering time for two pipes. The overhead is constant since it depends on the image size, not on the number of polygons.

Polygon count	131,000		5.6 Million	
Image size	512x512	1024x1024	512x512	1024x1024
Total time(secs)	7.709	15.276	136.846	140.756
Overhead(secs)				
communication	3.704	9.767	3.704	9.767
composite	1.397	2.381	1.397	2.381
total overhead	5.101	12.148	5.101	12.148
% of total time	66.2%	79.5%	3.72%	8.6%

Table 2. Overhead for binary-swap and compositing compared to total rendering time for four pipes. The overhead is constant since it depends on the image size, not on the number of polygons.

4 Conclusions

We have designed and implemented a multipipe parallel rendering system which takes advantage of multiple attached graphics adapters on high-end systems. For large polygon data sets, this method proves more interactive than utilizing

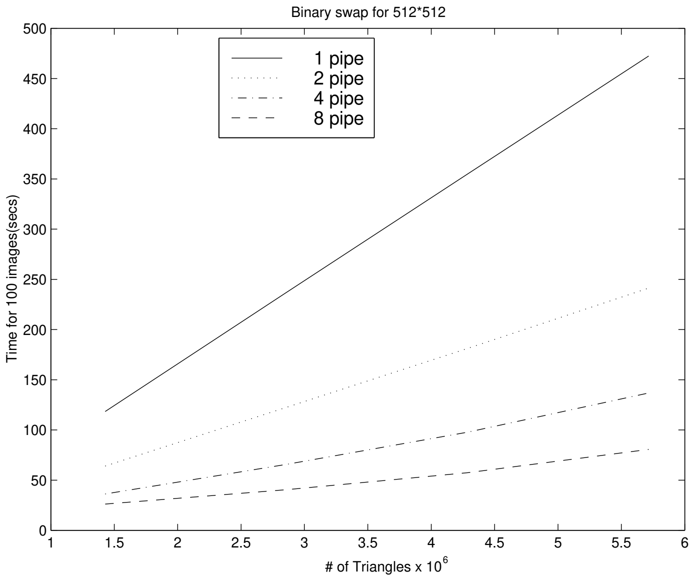

Fig. 4. Times for rendering a 512 x 512 image for 1.4 to 5.6 million polygons

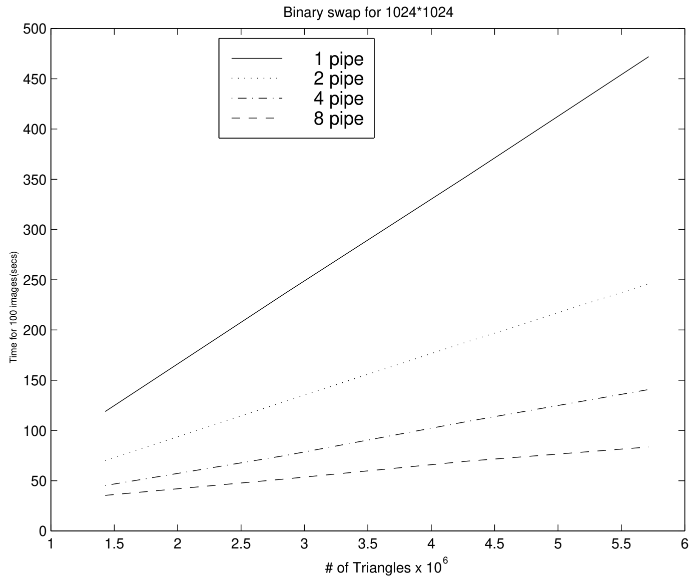

Fig. 5. Times for rendering a 1024 x 1024 image for 1.4 to 5.6 million polygons

a single graphics pipe. Surprisingly, the tradeoff for multi-pipe rendering occurs at less than 200k polygons for a 1024x1024 image. Given the large number of polygons extracted using typical isosurface methods, this clearly provides acceleration through the parallel rendering. We recognize that the cost is significantly higher yet many sites are taking delivery of such systems and we anticipate the usefulness of such parallel techniques particularly as data set sizes scale up. Since the cost of reading back the entire rendering window is fixed, based upon the Infinite Reality hardware, an enchancement would be to only read back the region of the image defined by the projection of the bounding box of the polygon set. Also, if it were possible to copy directly from graphics adapter to graphics adapter without caching the active images in shared memory, we would expect to see an increase in performance.

5 Acknowledgments

This work was supported by the University of Utah/SGI Visual Supercomputing Center, the Center for Simulation of Accidental Fires and Explosions, a DOE ASCI Level 1 Alliance Center, and the DOE Advanced Visualization Technology Center, a partnership between Univ of Utah, ANL, and LANL.

References

1. Kurt Akeley. RealityEngine graphics. *Computer Graphics*, 27:109–116, August 1993. ACM Siggraph '93 Conference Proceedings.
2. Kurt Akeley and Tom Jermoluk. High-performance polygon rendering. *Computer Graphics*, 22(4):239–246, August 1988. ACM Siggraph '88 Conference Proceedings.
3. David A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics and Applications*, 14(4), July 1994.
4. Kwan-Lui Ma et al. Parallel volume renderer using binary-swap image composition. *IEEE Computer Graphics and Applications*, 14(4), July 1994.
5. Steve Molnar et al. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4), July 1994.
6. C. Hansen and P. Hinker. Massively parallel isosurface extraction. In *Proceedings of Visualization '92*, pages 77–83, October 1992.
7. C. Hansen, M. Krogh, and W. White. Massively parallel visualization: Parallel rendering. In *Proceedings of SIAM Parallel Computation Conference*, February 1995.
8. William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. ACM Siggraph '87 Conference Proceedings.
9. Steven Molnar, John Eyles, and John Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics*, 26(2):231–240, July 1992. ACM Siggraph '92 Conference Proceedings.
10. John Montrym, Daniel Baum, David Dignam, and Christopher Migdal. Infinitereality: A real-time graphics system. *Computer Graphics*, 26:293–302, August 1997. ACM Siggraph '97 Conference Proceedings.
11. F. Ortega, C. Hansen, and J. Ahrens. Fast data parallel polygon rendering. In *Proceedings of Supercomputing '93*, pages 709–718, November 1993.