An Efficient Library for Parallel Ray Tracing and Animation

John Stone
Department of Computer Science
University of Missouri-Rolla
E-Mail: johns@cs.umr.edu

Abstract

Many simulations that are run on parallel computers can benefit from the use of parallel graphics libraries that can do in-place rendering. This paper describes a parallel rendering library that is being developed on the Intel iPSC/860, Paragon, and on other machines supported by the Message Passing Interface (MPI). The rendering library can be used standalone with a simple scene parser that is supplied, or as a rendering subprocess within a larger parallel computation. At the present time, the library supports a variety of modeling primitives including cylinders, patches, polygons, quadrics, spheres, and volumetric data sets. Features that set the ray tracing library apart from polygon based renderers are mirror reflection, refraction, shadows, texture mapping, and volume rendering. Current efforts are focused on improving volume rendering, overall performance, and ease-of-use.

1 Introduction

Ray tracing has been shown to be an effective technique for producing realistic synthetic images of three dimensional geometry. Ray tracing simulates rays of light as they interact with object surfaces. The simulation of surface physics allows ray tracing to render reflections, refractions, and shadows which look realistic. When compared with other rendering techniques such as polygon rendering, ray tracing offers enhanced realism at a higher computational cost [1]. The higher computational cost of ray tracing can be offset through exploitation of parallelsm. Ray tracing is well suited for rendering volumetric data such as MRI scans or fluid flow fields. Current acceleration techniques make ray tracing a feasible alternative to polygon rendering for many visualization applications.

The motivation for developing a parallel raytracing and animation library is to address the lack of easy-to-use, parallel rendering software. Many existing rendering libraries were designed to run on special purpose machines with polygon rendering hardware. Special purpose graphics hardware provides benefits when available, but detracts from the portability of applications which take maximal advantage of it. Dedicated rendering machines are hard to integrate closely with time marching simulation applications which execute on parallel computers. All data required for rendering each timestep of a simulation must be saved to files for batch rendering, or communicated to the rendering host at each time step. Parallel simulations can consume huge amounts of memory during execution, making data dumping at each time step impractical or impossible. This leads to the idea of rendering the data in place, without retransmission or dumping to disk. Parallel rendering libraries can be tightly integrated with applications so that rendering can take place on the same processors that are used for simulation. Crockett gives a good overview and discussion of the design considerations of parallel rendering libraries in [2].

Portability has been a major consideration in the design of this rendering library. In order to make the library portable to a wide range of machinery, it has been implemented using the Message Passing Interface [3]. Using MPI for message passing allows the same code to execute on the iPSC/860, Paragon, IBM SP-2, and on clusters of workstations. MPI **contexts** and **communicators** provide guaranteed-safe message handling when building applications with parallel libraries [4].

2 Library Features

2.1 Geometry

The goal of writing this rendering library is to provide the application programmer with a set of tools for realistic rendering of a wide range of geometry. The current implementation of the library supports rendering of several kinds of objects, including boxes, cylinders, smoothed triangles, flat triangles, quadric surfaces, rings, planes, spheres, and volumetric data. Currently, geometry is modeled solely through the use of the basic primitives. Future enhancements will allow matrix transformations to be applied to primitives to allow scaling and warping. Figure 1 shows the teapot generated by the SPD database using smoothed triangles [5]. SPD includes several test scenes which can be used when benchmarking rendering software. The image shown in Figure 2 is a starship rendered using a simple triangle mesh. Since many applications currently use polygon meshes to model objects, it is important to support a variety of polygon primitives. This is currently accomplished using triangle meshes. Future revisions of the rendering library will include a wider range of polygon rendering primitives. It is currently necessary to use a large number of primitives such as cylinders, spheres, and tri-

Figure 1: The SPD Teapot

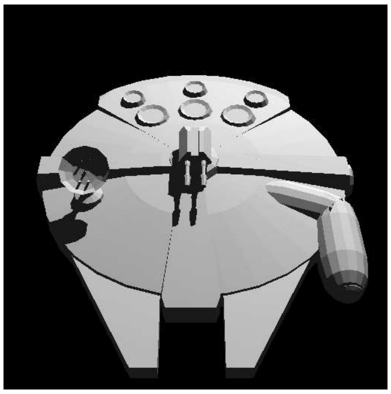

Figure 2: "Millenium Falcon" Object

angles to render complex models. In the near future the rendering library will allow creation of complex models using Constructive Solid Geometry (CSG) [1]. CSG uses unions, differences, and intersections of solids to define new solids. When used properly, CSG can drastically reduce the number of primitives required to effectively model an object. Objects that would ideally be modeled using CSG must currently be rendered using meshes of smoothed triangles.

2.2 Volume Rendering

In addition to geometric primitives such as spheres and triangles, the library can render volumetric data. Volume rendering can be used to generate images of MRI scans, clouds, fluid flow, and other grid or density data. The image shown in Figure 3 was generated from an MRI scan of a human head. The density values in the MRI scan were used to define opacity and color at each sample point. Ray casting is employed to sample the volume set and accumulate color and opacity as light passes through the volume. Since volume data values can be mapped to shading characteristics at run time, ray casting is a flexible method for rendering volume data. No preprocessing is necessary when rendering volume data with ray casting, all computations can be performed in place. The current implementation of the volume renderer handles scalar volume data only. Volume data can be supplied by the application, or loaded from disk files. Non-scalar vector fields from CFD simulations and similar applications will be supported soon.

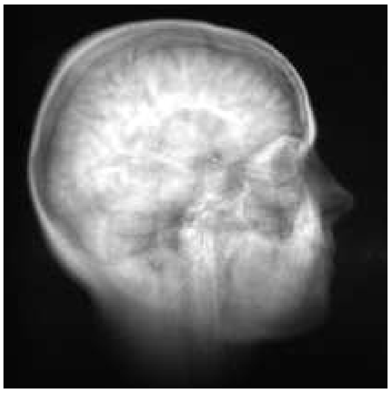

Figure 3: Volume Rendered MRI Scan

2.3 Textures

One of the strong points of the rendering library is its support for complex surface textures. In addition to color, reflection, and opacity, the rendering library can map complex procedural or image mapped textures onto objects. A number of procedural textures have been implemented including Noise based textures including marble. User supplied procedural textures aren't allowed yet, but a specification for user supplied procedural textures is forthcoming. Further work in procedural textures will focus on implementing hypertextures and clouds for volume rendering [6]. Images can be placed onto objects using a cylindrical, planar, or spherical mapping. Texture mapping is easy to use, and can be a powerful tool for increasing the realism of a scene.

3 Library Usage

It is easy to interface an application to the rendering library. The library uses simple data structures that are easy to work with. A single C header file is all that is needed to begin interfacing with the library. Applications make function calls to object and scene definition functions, which setup the object database and the scene parameters. The object definition functions use simple parameters, and take care of their own memory allocation. All objects must have a texture associated with them. In order to save memory, many objects may share the same texture. A simple example of object definition is the sphere. A sphere is defined by calling the **makesphere()** function. A call to **makesphere()** looks like this: **makesphere(texture, center, radius)**. The *texture, center,* and *radius* parameters describe the properties of the sphere that is added to the scene. Any number of objects may be defined until memory is exhausted. Once an object is defined, it is stored in an internal structure, and can't be modified. Animation is accomplished by resetting the renderer and rebuilding the scene for each frame. Volume data and image maps which are loaded from disk may be preserved between frames to avoid needless file I/O. At the present time object instancing isn't supported, future revisions of the library may add this feature. An in-depth guide to library usage is available by contacting the author.

4 Implementation Notes

The current library implementation uses a distributed memory MIMD model, regardless of the true underlying architecture. In order to keep processor utilization high, and communication low, the object database is replicated as among the processing elements. This approach works well for most scenes, since hundreds of thousands of the simple geometric primitives can be stored in the available node memory of most parallel computers. Although it performs well for simple objects, replication is a poor strategy for volumetric data and image map data. Work is in progress to distribute volume data and use ray passing when a ray pierces a volume with non-local data. Parallel CFD simulations already distribute grid data among processors so it is logical to render the data in place. The key factor in this strategy is to replicate or cache small memory size objects, and leave large memory size objects in place during rendering.

To exploit the data parallel nature of raytracing, the library assigns subsections of the image frame to seperate processors. The current implementation of the library uses static load balancing and scattered decomposition for the partitioning of the image plane. Since image blocks are computed in distributed memory, the final image must be constructed by writing the blocks into an output file. The library uses parallel file I/O to write pixel blocks to disk while computation proceeds. Since not all architectures support parallel I/O, message passing is used on networks of workstations and on the IBM SP-2. Future versions of the software will allow image data to be collected in memory on one node for interactive visualization or retransmission. An initial implementation of a runtime output window is already working on sequential machines.

All of the initial development and testing of the rendering library was conducted on a 32 node Intel iPSC/860 using NX. The current version of the rendering library uses MPI. Since the addition of MPI support, the library has been successfully tested on the iPSC/860, Paragon, IBM SP-2, SGI Challenge, and on networks of workstations. The current code includes makefile configuraions for each of the platforms above.

5 Results and Discussion

The static load balancing strategy used in the current implementation has been successfull on tests run to date. It achieves better than 92% efficiency in tests with up to 72 nodes on an Intel Paragon for the "balls" scene from the SPD package. The iPSC/860 and the Paragon both benefit from the concurrent I/O facilities (CFS and PFS) they have. The concurrent I/O performance advantage is quite noticable when rendering animations. Workstation clusters and the SP-2 suffer from the lack of parallel I/O. Parallel I/O helps the scalability of the rendering library significantly.

Performance testing has been conducted using the SPD database maintained by Eric Haines [5]. Table 1 shows the timing results for the "balls", "tetra", and "tree" scenes when run on a cluster of Suns and an Intel iPSC/860. The Suns used in the SPD timings were SparcStation 10 model 40 workstations running Solaris 2.3. The SPD results shown were performed according to the SPD guidelines, and include spatial subdivision preprocessing time as well as ray tracing time.

Table 1
SPD 3.6 Rendering Times (seconds)

Machine	Proc	Balls	Tetra	Tree
Sun Cluster	2	526	302	454
Sun Cluster	4	270	156	234
Sun Cluster	8	154	84	121
iPSC/860	8	210	121	211
iPSC/860	16	105	60	106
iPSC/860	32	53	31	55

The SPD results indicate that there is substantial room for improvement in the spatial subdivision implementation currently used. Results from previous tests show that only 75% to 90% of the ray intersection tests are being eliminated by the current subdivision scheme. Any improvement in the spatial subdivision performance will translate directly into a corresponding decrease in run time both sequentially and in parallel.

6 Future Work

Current shortcomings of the rendering library relate to overall rendering speed. This will be addressed through implementation of improved spatial subdivision techniques. The current technique uses simple recursive bounding boxes. Bounding slabs or one of the grid based subdivision methods should yield a speed improvement of 200% to 600% beyond the current level of performance [1]. In addition to speed improvements, additional functionality will be added in the areas of volume rendering, texturing, and modeling primitives. Short term goals include Hypertextures, user-defined procedural textures, CSG, and distributing volume data using ray passing.

References

[1] A. Glassner, *An Introduction to Ray Tracing.* Academic Press, 1989.

[2] T. Crockett, "Design considerations for parallel graphics libraries," in *Intel Supercomputer Users Group Conference*, 1994.

[3] M. P. I. Forum, "Mpi: A message-passing interface standard," Tech. Rep. Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[4] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI.* MIT Press, 1994.

[5] E. Haines, "A proposal for standard graphics environments," *IEEE Graph. Appl.*, November 1987.

[6] Ebert, Musgrave, Musgrave, Peachey, Perlin, and Worley, *Texturing and Modeling, A Procedural Approach.* Academic Press, 1994.