Interactive Ray Tracing

Steven Parker **William Martin** **Peter-Pike J. Sloan** **Peter Shirley** **Brian Smits** **Charles Hansen**

University of Utah, http://www.cs.utah.edu

Abstract

We examine a rendering system that interactively ray traces an image on a conventional multiprocessor. The implementation is "brute force" in that it explicitly traces rays through every screen pixel, yet pays careful attention to system resources for acceleration. The design of the system is described, along with issues related to material models, lighting and shadows, and frameless rendering. The system is demonstrated for several different types of input scenes.

CR Categories: 1.3.0 [Computer Graphics]: General; 1.3.6 [Computer Graphics]: Methodology and Techniques.

Keywords: Ray tracing, parallel systems, shading models

1 INTRODUCTION

Interactive rendering systems provide a powerful way to convey information, especially for complex environments. Until recently the only interactive rendering algorithms were hardware-accelerated polygonal renderers. This approach has limitations due to both the algorithms used and the tight coupling to the hardware. Software-only implementations are more easily modified and extended which enables experimentation with various rendering and interaction options.

This paper describes our explorations of an interactive ray tracing system designed for current multiprocessor machines. This system was initially developed to examine ray tracing's performance on a modem architecture. We were surprised at just how responsive the resulting system turned out to be. Although the system takes careful advantage of system resources, it is essentially a brute force implementation (Figure 1). We intentionally take the simple path wherever feasible at each step believing that neither limiting assumptions nor complex algorithms are needed for performance.

The ray tracing system is interactive in part because it runs on a high-end machine (SGI Origin 2000) with fast frame buffer, CPU set, and interconnect. The key advantages of ray tracing are:

- ray tracing scales well on tens to hundreds of processors;
- ray tracing's frame rendering time is sub-linear in the number of primitives for static scenes;
- ray tracing allows a wide range of primitives and user programmable shading effects.

1999 Symposium on Interactive 3D Graphics Atlanta GAUSA

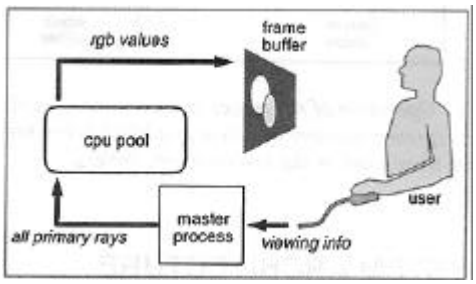

Figure 1: ***The ray tracing system discussed in this paper explicitly traces all rays on a pool of processors for a viewpoint interactively selected by the viewer.***

Figure 2: ***A portion of a 600 by 400 pixel image from our system running at approximately fifteen frames per second.***

The first item allows our implementation to be interactive, the second allows this interactivity to extend to relatively large (e.g. gigabyte) scenes, and the third allows the familiar ray traced look with shadows and specular reflection (Figure 2).

In the paper we stress the issues in ray tracing that change when we move from the static to the interactive case. These include achieving performance in synchronous or asynchronous (frameless) fashions (Section 2), and modifications to traditional Whitted-style lighting/shadowing model to improve appearance and performance (Section 3). We also discuss a few areas that might benefit from interactive ray tracing and show some of the environments we used in Section 4. We compare our work to the other work in parallel ray tracing in Section 5. We do not compare our work to the many object space methods available for simulating shadows and non-diffuse effects (e.g. Ofek and Rappoport [22]) which we believe comprise a different family of techniques. Our interactive implementation of ray tracing isosurfaces in trilinear volumes is described elsewhere [23].

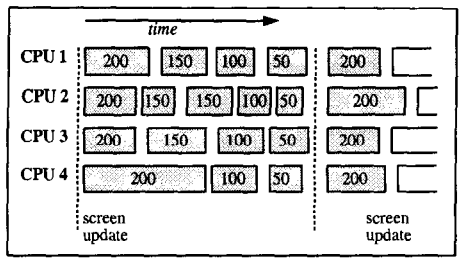

Figure 3: *Operation of ray tracer in synchronous mode. Numbers in boxes represent number of pixels in a block being processed. All pixels are traced before the screen swaps buffers.*

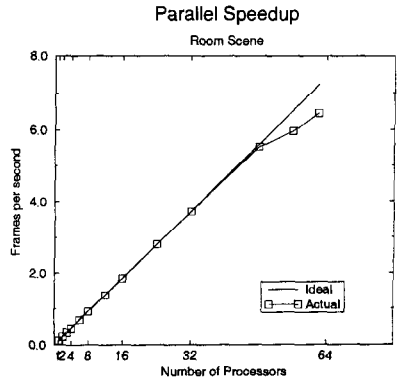

Figure 4: *Performance results for varying numbers of processors for a single view of the scene shown in Figure 16.*

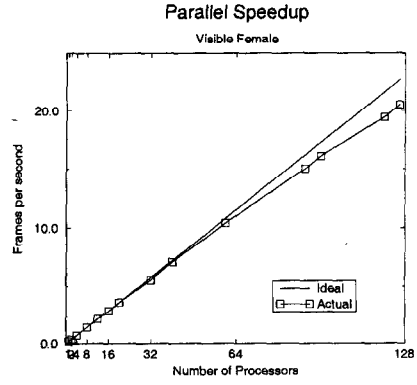

Figure 5: *Performance results for the visible female dataset, shown in Figure 9.*

2 SYSTEM ARCHITECTURE

It is well understood that ray tracing can be accelerated through two main techniques [26]: accelerating or eliminating ray/object intersection tests and parallelization. We employ both techniques in our system. We use a hybrid spatial subdivision which combines a grid based subdivision of the scene [10] with bounding volumes [17]. For a given scene, we can empirically test both methods to arrive at the 'best' combination where 'best' is dependent upon the scene geometry and the particular application. The beauty of the interactive system is the ability to rapidly explore tradeoffs such as different spatial subdivision techniques.

Ray tracing naturally lends itself towards parallel implementations. The computation for each pixel is independent of all other pixels, and the data structures used for casting rays are usually read-only. These properties have resulted in many parallel ray tracers, as discussed in Section 5. The simplest parallel shared memory implementation with reasonable performance uses Master/Slave demand driven scheduling as follows:

Master Task
initialize model
initialize ray tracing slaves on each free CPU
loop
  update viewing information
  lock queue
  place all primary rays in queue
  unlock queue
  when the queue is empty redraw screen and handle user input
end loop

The ray tracing slaves are simple programs that grab primary rays from the queue and compute pixel RGB values:

Slave Task
initialize memory
loop
  if *queue is not empty* ***then***
    lock queue
    pop ray request
    unlock queue
    compute RGB for pixel
    write RGB into frame buffer pixel
  end if
end loop

This implementation would work, but it would have excessive synchronization overhead because each pixel is an independent task. The actual implementation uses a larger basic task size and runs in conventional or frameless mode as discussed in the next two sections.

2.1 Conventional Operation

To reduce synchronization overhead we can assign groups of rays to each processor. The larger these groups are, the less synchronization is required. However, as they become larger, more time is potentially lost due to poor load balancing because all processors must wait for the last job of the frame to finish before starting the next frame. We address this through a load balancing scheme that uses a static set of variable size jobs that are dispatched in a queue where jobs linearly decrease in size. This is shown in Figure 3.

Figure 3 has several exaggerations in scale to make it more obvious. First, the time between job runs for a processor is smaller than is shown in the form of gaps between boxes. Second, the actual jobs are multiples of the finest tile granularity which is a 128 pixel tile (32 by 4). We chose this size for two reasons: cache coherency for the pixels and data cache coherency for the scene. The first reason is dictated by the machine architecture which uses 128 byte cache lines (32 4-byte pixels). With a minimum task granularity of a cache line, false sharing between image tiles is eliminated. A further advantage of using a tile is data cache reuse for the scene geometry. Since primary rays exhibit good spatial coherence, our system takes advantage of this with the 32 by 4 pixel tile.

The implementation of the work queue assignment uses the hardware fetch and op counters on the Origin architecture. This allows efficient access to the central work queue resource. This approach to dividing the work between processors seems to scale very well. In Figure 4 we show the scalability for the room scene shown in Figure 16. We used up to 64 processor (all that are available lo-

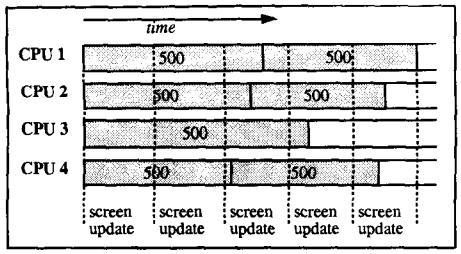

Figure 6: *Operation of ray tracer in asynchronous (frameless) mode. Screen is constantly updating and each processor is repeatedly tracing its set of pixels.*

cally) and found that up through about 48 we achieved almost ideal performance. Above 48 there is a slight drop off. We also show performance data for interactively ray tracing the iso-surfaces of the visible female dataset in Figure 5. For this data we had access to a 128 processor machine and found nearly ideal speed ups for up to 128 processors.

Since most scenes fit within the secondary cache of the processor (4 Mb), the memory bandwidth used is very small. The room scene, shown in Figure 4 uses an average of 9.4 Mb/s of main memory bandwidth per processor. Ironically, rendering a scene with a much larger memory footprint (rendering of isosurfaces from the visible female dataset [23]) uses only 2.1 to 8.4 Mb/s of main memory bandwidth. These statistics were gathered using the SGI perfex utility, benchmarked with 60 processors.

Since ray tracing is an inherently parallel algorithm, efficient scaling is limited by only two factors: Load balance and synchronization. The dynamic work assignment scheme described earlier is used to limit the effect of load imbalance. Synchronization for each frame can limit scaling due to the overhead of the barrier. The standard barrier provided in Irix requires an average of 5 milliseconds to synchronize 64 processors, which limits the scaling at high framerates. An efficient barrier was implemented using the "fetchop" atomic fetch-and-op facilities in the Origin. A barrier operation consumes 61 microseconds on average, which is an insignificant percentage of the frame time.

2.2 Frameless Rendering

For frameless rendering [3, 7, 36] the viewpoint and screen are updated synchronously, but the pixels are updated according to an asynchronous quasi-random pattern. Our implementation for this is summarized in Figure 6.

The implementation assigns a static pixel distribution to the rendering threads - every processor has a list of pixels that it will update, requiring minimal synchronization between threads. The rendering thread handles user input and draws the buffer to the screen at regular intervals. This is done asynchronously to the rendering threads. The rendering threads periodically update their camera - this is done at a specified rate expressed as a percentage of the pixels that thread owns. The display thread is modified so that it updates the screen at some user defined frame rate.

When creating a "static" pixel distribution (partitioning the screen between processors), there are two conflicting goals: 1) maintain coherent memory access; 2) have a more random distribution (incoherent memory) of pixels. The first is important for raw system efficiency, and the second is important to avoid visually distracting structure during updates.

In the current system we partition the image plane using a Hilbert curve (this maps the image to a 1D line), and then break this line into "chunks", these chunks are distributed to the processors in a round robin fashion (processors interleaved with chunk granularity along the 1D domain of the Hilbert curve). Each thread then randomly permutes its chunks so that the update doesn't always exactly track the Hilbert curve.

When updating the image, pixels can be blended into the frame buffer. This causes samples to have an exponential decay and creates a smoother image in space and time. We can use jittered sampling where there are four potential sample locations per pixel and two of them are updated when the pixel is updates, so the pixel is only fully updated after two passes. This implements the "frameless anti-aliasing" concept of Scher Zagier [35].

One nice property of a static pixel distribution is the ease of keeping extra information around (each thread just stores it - and no other threads will access this memory.) This can be used for computing a running average, sub pixel offsets for jittered sampling, a running variance computation or other information about the scene associated with that pixel (velocity, object ids, etc.).

3 IMPLEMENTATION DETAILS

Users of traditional ray tracers feel free to change the lighting and material parameters when the viewpoint is changed, and to add multiple lights to achieve a desired lighting effect. These are not practical in an interactive ray tracer where the lighting and material parameters are static as the viewpoint changes, and where even one light is expensive in terms of framerate. To help reduce the need for such traditional hacks, our implementation modifies the traditional key components of a ray tracer: lighting, material models, shadows, and ray-object intersection routines. In Section 3.1 we discuss how we handle and modify material models and lighting in a dynamic context. In Section 3.2 we discuss how we approximate soft shadows efficiently. In Section 3.3 we discuss how we compute ray-object intersections for spline surfaces.

3.1 Lighting and Materials

The traditional "Whitted-style" illumination model has many variations, but for one light the following formula is representative:

$$L = k_d \left(l_a + s l_e \hat{\mathbf{n}} \cdot \hat{\mathbf{l}} \right) + s l_e k_h \left(\hat{\mathbf{h}} \cdot \hat{\mathbf{l}} \right)^N + k_s L_s + k_t L_t, \quad (1)$$

where the vector quantities are shown in Figure 7, and L is the radiance (color) being computed, s is a shadow term that is either zero or one depending on whether the point luminaire is visible, k_d is the diffuse reflectance, l_a is the ambient illumination, l_e is the luminaire color, k_h is the Phong highlight reflectance, k_s is the specular reflectance, L_s is the radiance coming from the specular direction, k_t is the specular transmittance, and L_t is the radiance coming from the transmitted direction. Although this basic formula serves us well, we believe some alterations can improve performance and appearance. In particular, we are careful in allowing k_s and k_t to change with incident angle, we modify the ambient component l_a to be a very crude approximation to global illumination (Section 3.1.1), and we allow soft shadowing by making s vary continuously between zero and one (Section 3.2). Finally, we break the materials into several classes to compute only non-zero coefficients for efficiency.

One well-known problem with Equation 1 is that the specular terms do not change with incident angle. This is different from the behavior of materials in the real world [14]. In a conventional ray tracer the values of k_d, k_s and k_t can be hand-tuned to depend on viewpoint but in an interactive setting this does not work well. Instead, we first break down materials into a few distinct subjective categories suggested in [31]: *diffuse, dielectric, metal,* and *polished.* The modifications for these materials is described below:

Diffuse. For diffuse surfaces we use Equation 1 with $k_h = k_s = k_t = 0$. This is the same as a conventional ray tracer.

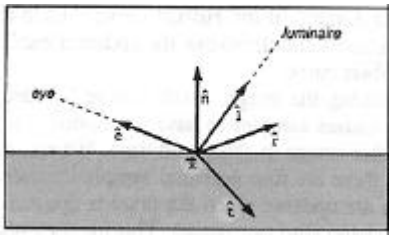

Figure *7: The directional quantities associated with Equation 1.*

Metal. Metal has a reflectance that varies with incident angle [6]. We are currently ignoring this effect, and other effects of real metal, and using traditional Whitted-style lighting. We use Equation 1 with $k_d = k_t = 0$, and $k_h = k_s$.

Dielectric. Dielectrics, such as glass and water, have reflectances that depend on viewing angle. These reflectances are modeled by the Fresnel Equations, which for the unpolarized case can be approximated by a polynomial developed by Schlick [28]:

$$k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}) = R_0 + (1 - R_0)(1 - \hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^5,$$

and k_t is determined by conservation of energy:

$$k_t(\hat{\mathbf{e}}, \hat{\mathbf{n}}) = 1 - k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}).$$

The internal attenuation of intensity I is the standard exponential decay with distance t according to extinction coefficient κ: I(t) = $I(0)exp(-\kappa t)$. To approximate the specular reflection of an area light source we add a Phong term to dielectrics as well.

Polished. We use the coupled model presented in [30]. This model allows the k_s to vary with incident angle, and allows the diffuse appearance to decrease with angle. As originally presented, it is a BRDF, but it is modified here to be appropriate for a clamped RGB lighting model with an ambient component:

$$\begin{aligned} L \;=\; & k_d l_a (1 - k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}})) + \\ & s k_d l_e \left(1 - (1 - \hat{\mathbf{n}} \cdot \hat{\mathbf{e}})^5\right) \left(1 - (1 - \hat{\mathbf{n}} \cdot \hat{\mathbf{l}})^5\right) + \\ & s k_h (\hat{\mathbf{h}} \cdot \hat{\mathbf{l}})^N + \\ & k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}) L_s, \end{aligned} \tag{2}$$

where the first term assumes the ambient component arises from directionally uniform illumination.

3.1.1 Ambient Lighting

The ambient term l_a in Equation 1 is a crude approximation used in conventional ray tracers to avoid computing an indirect lighting term. It is not meant to be physically accurate, but instead to illuminate those areas that are not directly lit by the luminaires. Given this, its main failing is that the uniform intensity causes diffuse objects to appear flat when the surface faces away from the light source. One way to avoid this is to put a fill-light at the eye point, but we feel this is distracting for a moving viewpoint. An alternative is to allow the ambient coefficient to vary with position $\vec{\mathbf{p}}$ and orientation: $l_a(\vec{\mathbf{p}}, \hat{\mathbf{n}})$. This can be a simple heuristic, or based on radiosity solutions [13]. Our motivation for using a more sophisticated ambient term is to allow shading variation on surfaces that are not directly lit without the computational cost of adding additional lights. This can be accomplished by assuming the ambient term arises due to illumination from a background divided evenly between two intensities A and B (Figure 8). The angle θ will vary from zero to π radians. For $\theta = 0$ the surface will only "see" ***A so*** the ambient term will be ***A.*** As θ increases the ambient term will

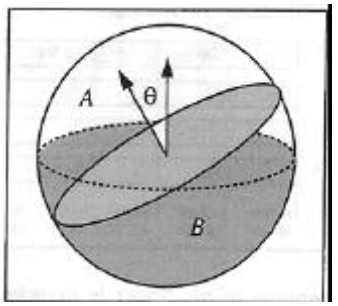

Figure *8: A surface is illuminated by a hemisphere with colors A and B.*

gradually change to $(A + B)/2$ at $\theta = \pi/2$, and finally change to B as the surface fully faces the bottom hemisphere. Nusselt's analog [4] allows us to derive the full relationship:

$$l_a(\theta) = \begin{cases} \left(1 - \frac{\sin\theta}{2}\right) A + \left(\frac{\sin\theta}{2}\right) B & \text{if } \theta < \frac{\pi}{2}, \\ \left(\frac{\sin\theta}{2}\right) A + \left(1 - \frac{\sin\theta}{2}\right) B & \text{otherwise.} \end{cases}$$

Either the user can set A and B algorithmically or by hand. We set ours by hand, but some heuristics can aid in selection. If we envision the hemisphere-pair as approximating indirect lighting of an object in a room, then the "walls" opposite the light are well illuminated and bright. So the hemisphere can be roughly aligned to the light source, with the hemisphere in the direction of the light source darker than the one pointing away from the light source. As advocated by Gooch et al. [12], we can accent the shape using a cool-to-warm color shift by making sure the light source is yellow (warm) and the hemisphere facing away from the light is blue (cool). Our ambient approximation is shown in Figure 9 and is not measurably slower than a constant ambient component for non-trivial mmodels.

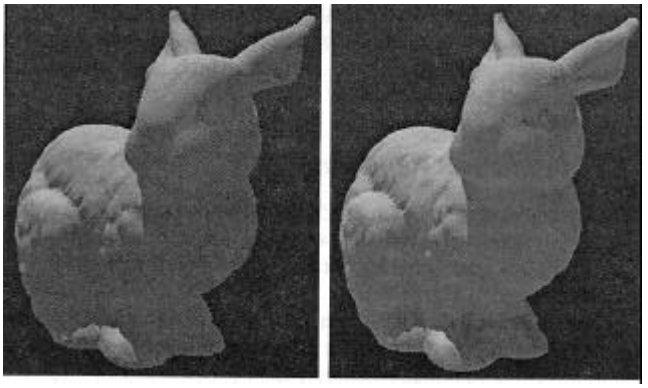

Figure *9: Left: simple ambient approximation. Right: directionally varying ambient approximation.*

3.2 Shadows

One of the limitations of ray tracing is the hard edges computed for shadows. In addition to aesthetic reasons, there is evidence that soft edged shadows aid in accurate spatial perception [19]. Ray tracing methods that produce accurate soft shadows such as ray tracing with cones [1] or probabilistic ray tracing [5] stress accurate soft shadows, but dramatically increase computation time relative to hard shadow computation. In this section we examine how to compute soft-edged shadows approximately so that interactivity can be maintained.

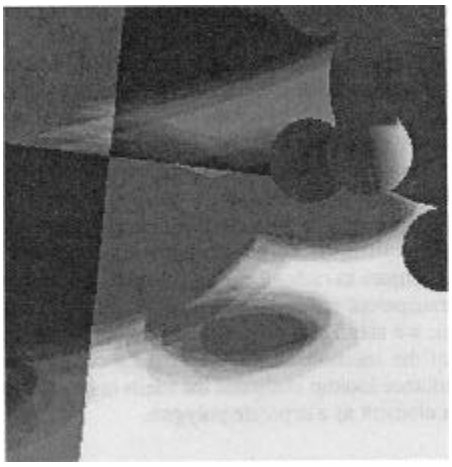

Figure 10: *A beam traced shadow with five samples. Note that there are discontinuities within the shadow.*

One option to improve performance is to do explicit multi-sampling of the light with a "beam" made up of a small number of rays [15]. Since the number of rays is small, there will be visual artifacts but interactive performance will be possible (Figure 10). To speed up this computation we can precompute the rays in the beam if we assume the luminaire is far away, and we can vectorize the intersection computation against each geometric primitive. This is similar to traversing efficiency structures using bundles rather than single rays [9]. Although this optimization gives us a factor of two performance over the unvectorized version, it is still too slow for many shadow rays.

An alternative to computing accurate soft shadows is to soften the edges of hard shadows. This is essentially the technique used in depth buffer algorithms [25] where the binary shadow raster can be filtered. However we want to simulate the change in penumbra width we see in real shadows. Such an effect requires more sophisticated filtering. This means shadow penumbra width should behave in a believable way, starting at zero at the occluder and increasing linearly with distance from the occluder.

It is hard for observers to tell the difference between shadows cast by differently shaped lights. For this reason we assume spherical lights. We do a rough calculation at each illuminated point of what fraction s of the light is visible, and attenuate the unshadowed illumination by s. Thus our goal is to estimate s in efficiently and to visually plausible results.

Rather than creating a correct shadow created by an area source, the algorithm creates a shadow of a ***soft-edged*** *object* from a point source (Figure 11). The penumbra is the shadow of the semi-opaque (outer) object that is not also shadowed by the opaque (inner) object. The transparency of the outer object increases from no transparency at the inner object to full transparency at the boundary of the outer object. For an isolated object, we can use inner and outer offsets of the real object to achieve believable results. We also need to make the intensity gradient in the penumbra natural. This can be achieved by computing the shadowing variable s beginning at $s = 0$ on the penumbra/umbra boundary (the surface of the inner object) and increasing non-linearly with distance to $s = 1$ on the outer boundary of the penumbra (the surface of the outer object).

The above approach will give an approximate soft shadow. The size of the penumbra is based on the size of the offsets used to create the inner and outer objects. In order to have the penumbra width change plausibly, the offsets need to change based on the distance along the shadow ray and the size of the light source, as illustrated in Figure 12. This requires modifying the intersection tests

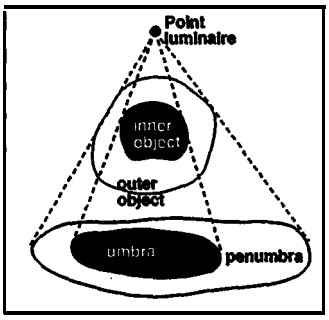

Figure 11: ***The inner object is opaque and the outer object's opacity falls off toward its outer boundary.***

for shadow rays. The details of this approach, including solutions to light leaking between two objects and the intersection tests and bounding box construction for polygons and spheres with varying offsets are discussed in more depth by Parker *et al. [24]. The* results are shown in Figure 13.

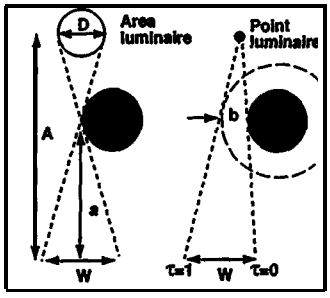

Figure 12: ***Choosing the size of the outer objectfor a given configuration.***

3.3 Spline Surfaces

In most traditional rendering systems NURBS are tessellated, often outside the graphics API in order to have more control over the accuracy. This can lead to an explosion in the amount of data that needs to be stored and then sent down the rendering pipeline. Ray tracing does not have this limitation.

Intersection tests with NURBS have been done in several ways (e.g., [16, 29, 33]. Our approach computes an estimate to the intersection point and then uses a brute force approach to compute

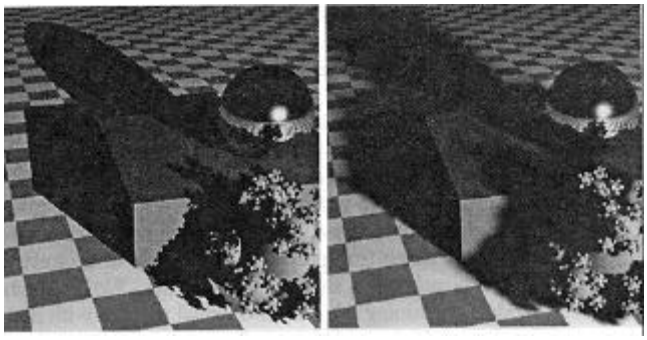

Figure 13: ***Left: one sample per pixel with hard shadows. Right: one sample per pixel with soft shadows. Note that the method captures the singularity near the box edge.***

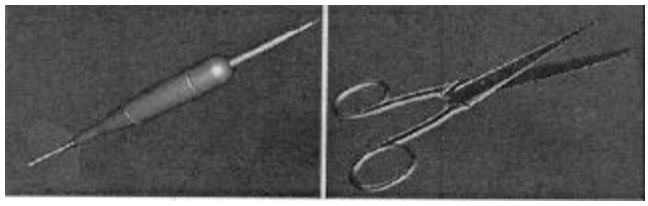

Figure 14: ***Two images rendered directly from the spline model.***

the actual intersection point. Surface parameter spaces are subdivided to a user-specified depth, and the quadtree that results is used to construct an inn-a-surface bounding volume hierarchy. Axis-aligned bounding volumes are used to preserve consistency with the overall infrastructure. Bounding volume hierarchies are built bottom up. It is important to note that this will result in tighter volumes than top down construction (since the subdivided control meshes converge to the surface).

Intersections with the leaf nodes of the bounding volume tree are computed using Broyden's method. This is a pseudo-Newton solver which approximates the value of the Jacobian. It converges more slowly than Newton, but requires fewer function evaluations. The initial guess is given by the average of the boundary parameter values of the patch in question. Patches are allowed to overlap by a small percentage of their parametric domains, thereby lessening the chance of cracks.

Usually fewer than three iterations of the root finder are required to converge to a suitably refined surface. The cost of storage is one copy of the original control mesh, and for each leaf node in the intra-surface bounding volume hierarchy, four doubles denoting the parametric interval it covers. In addition, we require each processor to reserve a scratch area so that the spline basis functions can be computed without needing to lock the data. The cost of this storage is $m+n$ where m and n are the maximum control mesh dimensions over all surfaces in the scene.

4 RESULTS

The final rendering system is interactive on relatively few (8) processors, and approaches real time for complex environments on 64 or more processors. It runs well on a variety of models from different application areas. Its flexibility allows several different display modes, all of which are applicable to 'the various models.

Ray tracing is ideal for showing dynamic effects such as specular highlights and shadows. Dynamic objects are more difficult to incorporate into a ray tracer than into a z-buffer algorithm as current acceleration schemes are not dynamic [11]. Our current workaround is to keep dynamic objects outside the acceleration scheme and check them individually for each ray. Obviously this only works for limited numbers of dynamic objects. In Figure 2 we show a static image from a set of bouncing balls using the soft shadow approximation.

Computer-aided design usually uses both curved surfaces and non-diffuse objects, such as a windshield made from glass. Ray tracing can render curved surfaces directly, making it ideal for spline models. The ability to calculate accurate reflections across the surface make is possible to evaluate the smoothness and curvature of the models for aesthetic purposes. A sample of a directly ray traced spline primitives is shown in Figure 14. We have run on several models containing 20-2000 individual patches with runtimes ranging from 1-20 fps at 512 by 512 pixels on 60 processors.

Ray tracing time is sub-linear with respect to model size. This allows us to interact with very large models. One area that creates large models is scientific visualization. In Figure 15 we show a visualization of a stress simulation. Each node in the simulation is represented by a sphere. There are 35 million spheres in this model. Unlike conventional rendering systems, the high depth complexity has very little effect on the rendering times. Another area that can create complex models is architectural design. The model in Figure 16 contains roughly 75,000 polygons and a spline teapot. An area we would like to explore is the use of interactive ray tracing for walk throughs of globally illuminated static environments, where the illumination information has been computed in advance by such techniques as radiosity or density estimation. Usually specular and transparent effects are missing from such walk throughs. In addition, we should be able to easily allow higher order reconstruction of the solution. Also, we could greatly reduce polygon count if radiance lookup evaluates the mesh instead of representing each mesh element as a separate polygon.

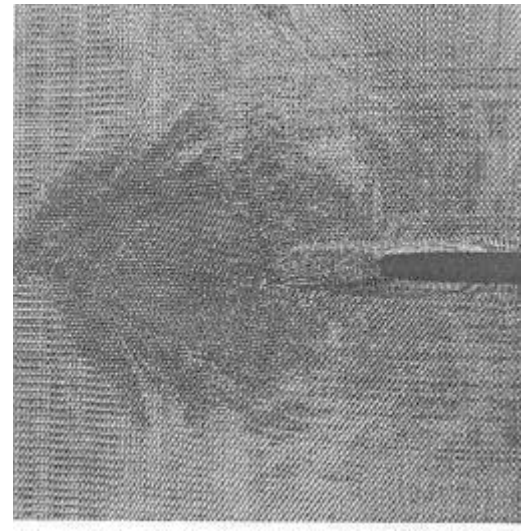

Figure 15: ***Simulation of crack propagation visualized using 35M spheres. This image at 512 by 512 pixels runs approximately 15 frames per second on 60 CPUs.***

5 RELATED WORK

Ray tracing has long been a focus for acceleration through parallel techniques. There are two general parallel methods which are used for such acceleration: demand scheduling and data parallel. Demand driven scheduling refers to distributing tasks upon demand. Data parallel methods partition the data and assign tasks to the processors which contain the required data. Hybrid methods combine these two paradigms generally by partitioning the tasks into those requiring a small amount of data and those requiring a large amount. Since shared memory processors with large amounts of memory have only recently been commercially available, most parallel ray tracing research has focused on distributed memory architectures. For shared memory parallel computers, demand driven scheduling methods tend to lead to the best performance [26]. Our implementation is based on demand driven scheduling where the task granularity is rendering an 32 by 4 pixel tile. In this section, we provide a comparison with several related parallel ray tracing implementations. A more thorough general review is provided by Reinhard and Jansen [26].

Muuss and researchers from ARL have experimented with parallel and distributed ray tracing for over a decade [20, 21]. In their recent work, they describe a parallel and distributed real-time ray

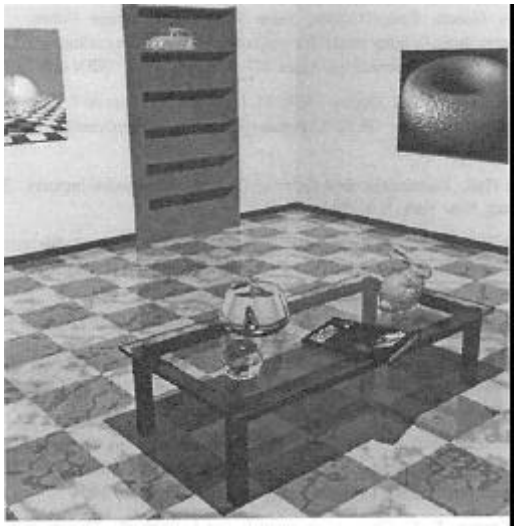

Figure 16: A ***model with splines, glass, image textures, and procedural solid textures. At 512 by 512 pixels this image is generated at approximately 4 frames per second on 60 CPUs.***

tracer running on a cluster of SGI Power Challenge machines [21]. One of the differences between BRL's effort and ours is the geometric primitives used. Their geometry is defined by through a CSG modeler, BRL-CAD. Additionally, we leverage the tight coupling of the graphics hardware on the SGI Origin while their system uses an image decomposition scheme combined with a network attached framebuffer. Muuss points out that synchronization, particularly at the frame level, is critical for real-time ray tracing [21]. Our research indicates that synchronization within a frame is also critical as noted by our dynamic load balancing scheme. Although not reported in the literature, ARL's current effort seems to have a comparable framerate as ours (Muuss, personal communication at SIGGRAPH98).

Keates and Hubbold use a distributed shared memory architecture, the KSRl, to implement a demand driven ray tracer which renders a simple scene is slightly over 1.8 seconds for 256 processors [18]. Their implementation is similar to ours in that they use the brute force technique of parallelizing over rays. However, their work differs in the granularity of work distribution, the method used for load balancing, and results based upon architecture. Their implementation split the screen into regions and divided the work among the CPUs. It is not clear how large the regions were but one is lead to believe the regions are larger than the 32 pixel regions used in our implementation. They report problems with load balancing and synchronization. They address these by a two level hierarchy for screen space subdivision similar to Ellsworth [8]. Our system uses a different strategy for load balancing of decreasing granularity of assigned work which empirically yields better results. This also assists in synchronization which is why this issue has not been a problem for us.

Singh et al. reported on using the Stanford DASH distributed shared memory machine for ray tracing [32]. Their implementation used an image decomposition scheme which subdivided the image among the available processors. Within a processor, the sub-image a further subdivided into 8 by 8 pixel tiles. As in our system, their implementation noted the advantage of data cache reuse for object intersection test. Their work differed from ours in the load balancing scheme. They used task stealing rather than demand driven scheduling. We find that the simpler approach of using a task queue with good dynamic load balancing provides excellent results without the complexity of performing task stealing. The fetch and op hardware in the Origin architecture allows the task queue to perform well even on a large number of processors.

Yoon et al. use an image partitioning scheme which statically load balances the tasks by interleaving pixels and distributing among nodes the scene data while replicating the spatial hierarchy on each node [34]. Their work attempts to prefetch data for each ray task. Their work differs from ours in two major respects: load balancing and machine architecture. Our implementation effectively exploits dynamic load balancing through the heuristic of decreasing task size while Yoon et al. employ static load balancing through pixel assignment. Since their work focuses on a distributed memory architecture, they need to explicitly address data distribution while our implementation exploits the CC-NUMA distributed shared memory.

Reinhard and Jansen use a hybrid scheduling method for parallel ray tracing [27]. Their implementation divides the ray tracing task into those tasks which require limited amounts of data and those that require more substantial amounts of data. Since their spatial subdivision hierarchy, but not the leaf nodes, is replicated on each processor, tasks using these are demand scheduled whereas tracing rays through the objects within the leaf nodes is performed in a data parallel fashion. Their method makes novel use of this combined scheduling scheme which provides better performance on distributed memory parallel computers. Since our method exploits the distributed shared memory architecture, we can achieve very good performance with only demand scheduling.

Bala et al. describe a bounded error method for ray tracing [2]. For each object surface, their method uses a 4D linetree to store a collection of interpolants representing the radiance from that surface. If available, these interpolants are reprojected when the user's viewpoint changes. If not, the system intersects the ray with the scene checking for a valid interpolant at the intersection point. If one is found, the radiance for that pixel is interpolated. Otherwise, using that linetree cell, an attempt is made to build an interpolant. If this is within an error predicate, it is used otherwise the linetree cell is subdivided and the system falls back to shading using a standard ray tracing technique. The acceleration is based upon the utilization of previously shaded samples bounded by an error predicate rather than fully tracing every ray. Our system is brute-force and traces every ray in parallel. Bala's method is oriented toward a more informed and less parallel strategy, and is currently not interactive. Moving objects would pose a problem for the linetree based system whereas they can be handled in our implementation. Using reprojection techniques might further accelerate our system.

6 CONCLUSION

Interactive ray tracing is a viable approach with high end parallel machines. As parallel architectures become more efficient and cheaper this approach could have much more widespread application. Ray tracing presents a new set of display options and tradeoffs for interactive display, such as soft shadows, frameless rendering, more sophisticated lighting, and different shading models. The software implementation allows us to easily explore these options and to evaluate their impact for an interactive display.

We believe the following possibilities are worth investigating:

- How should antialiasing be handled?
- How do we handle complex dynamic environments?
- How do we ensure predictable performance for simulation applications?
- What should the API be for an interactive ray tracer?

. How could an inexpensive architecture be built to do interactive ray tracing?

The first three items above highlight significant limitations of our current system: antialiasing is brute-force and thus too costly, and performance can be slow or unpredictable because there is a complex interaction between efficiency stucture build time, traversal time, and view-dependent performance. How much of these are due to the batch nature of traditional ray tracing methodology versus intrinsic limitations is not yet clear.

Additionally, we feel that an interactive ray tracer can help answer more general questions in interactive rendering, such as:

- How important are soft shadows and indirect illumination to scene comprehension and how accurate do they need to be?
- Are more physically accurate BRDF's more or less important in an interactive setting?
- Do accurate reflections give significant information about surface curvature/smoothness?

The ability to have more complete control over these features allows us to investigate their effects more completely.

Acknowledgments

Thanks to Chris Johnson for providing the open collaborative research environment that allowed this work to happen. Thanks to Elaine Cohen for many conversations on spline surface intersection and to Amy Gooch for help with model acquisition and conversion. Thanks to Dave Beazley and Peter Lomdahl at Los Alamos National Laboratory for the 35 million sphere data set. Special thanks to Jamie Painter and the Advanced Computing Laboratory at Los Alamos National Laboratory for access to a 128 processor machine for final benchmarks. The bunny model appeared courtesy of the Stanford University Graphics Laboratory. This work was supported by the SGI Visual Supercomputing Center, the Utah State Centers of Excellence, the Department of Energy and the National Science Foundation.

References

[1] J. Amanatides. Ray tracing with cones. *Computer Graphics*, pages 129–135, July 1984. ACM Siggraph '84 Conference Proceedings.

[2] Kavita Bala, Julie Dorsey, and Seth Teller. Bounded-error interactive ray tracing. Technical Report LCS TR-748, MIT Computer Graphics Group, March 1998.

[3] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: Double buffering considered harmful. *Computer Graphics*, 28(3):175–176, July 1994. ACM Siggraph '94 Conference Proceedings.

[4] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Boston, MA, 1993.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(4):165–174, July 1984. ACM Siggraph '84 Conference Proceedings.

[6] Robert L. Cook and Kennneth E. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3):307–316, August 1981. ACM Siggraph '81 Conference Proceedings.

[7] Robert A. Cross. Interactive realism for visualization using ray tracing. In *Proceedings Visualization '95*, pages 19–25, 1995.

[8] David A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics and Applications*, 14(4), July 1994.

[9] Bernd Fröhlich. *Ray Tracing mit Strahlenbündeln (Ray Tracing with Bundles of Rays)*. PhD thesis, Technical University of Braunschweig, Germany, 1993.

[10] Akira Fujimoto, Takayu Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, pages 16–26, April 1986.

[11] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[12] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH 98 Conference Proceedings*, pages 447–452, July 1998. ISBN 0-89791-999-8.

[13] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics & Applications*, 18(2):32–43, March–April 1998.

[14] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, N.Y., 1988.

[15] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, July 1984.

[16] James T. Kajiya. Ray tracing parametric patches. In *SIGGRAPH '82*, 1992.

[17] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986. ACM Siggraph '86 Conference Proceedings.

[18] M.J. Keates and R.J. Hubbold. Accelerated ray tracing on the KRS1 virtual shared-memory parallel computer. Technical Report UMCS-94-2-2, Computer Science Department, University of Manchester, February 1994.

[19] D. Kersten, D. C. Knill, Mamassian P, and I. Bülthoff. Illusory motion from shadows. *Nature*, 379:31, 1996.

[20] Michael J. Muuss. Rt and remrt - shared memory parllel and network distributed ray-tracing programs. In *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, October 1987.

[21] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, June 1995.

[22] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *SIGGRAPH 98 Conference Proceedings*, pages 333–342, July 1998.

[23] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings Visualization '98*, 1998.

[24] Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, October 1998. http://www.cs.utah.edu/~bes/papers/coneShadow.

[25] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, 21(4):283–292, July 1987. ACM Siggraph '87 Conference Proceedings.

[26] E. Reinhard, A.G. Chalmers, and F.W. Jansen. Overview of parallel photorealistic graphics. In *Eurographics '98*, 1998.

[27] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7), July 1997.

[28] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 73–84, June 1993.

[29] Thomas W. Sederberg and Scott R. Parry. Comparison of three curve intersection algorithms. *Computer-aided Design*, 18(1), January/February 1986.

[30] Peter Shirley, Helen Hu, Brian Smits, and Eric Lafortune. A practitioners' assessment of light reflection models. In *Pacific Graphics*, pages 40–49, October 1997.

[31] Peter Shirley, Kelvin Sung, and William Brown. A ray tracing framework for global illumination systems. In *Proceedings of Graphics Interface '91*, pages 117–128, June 1991.

[32] J.S. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7), July 1994.

[33] Wolfgang Stürzlinger. Ray-tracing triangular trimmed free-form surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3), July-September 1998.

[34] H.J. Yoon, S. Eun, and J.W. Cho. Image parallel ray tracing using static load balancing and data prefetching. *Parallel Computing*, 23(7), July 1997.

[35] Ellen Scher Zagier. Frameless antialiasing. Technical Report TR95-026, UNC-CS, May 1995.

[36] Ellen Scher Zagier. Defining and refining frameless rendering. Technical Report TR97-008, UNC-CS, July 1997.

Figure A: *A portion of a 600 by 400 pixel image from our system running at approximately fifteen frames per second.*

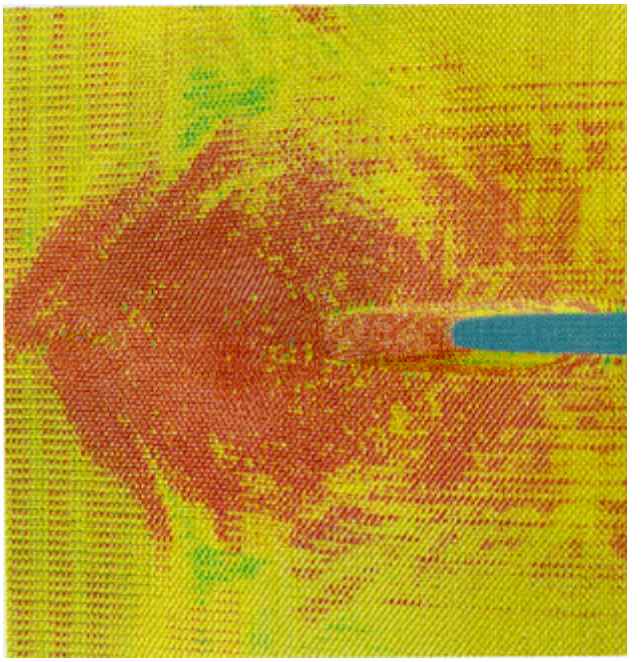

Figure C: *Simulation of crack propagation visualized using 35M spheres. This image at 512 by 512 pixels runs approximately 15 frames per second on 60 CPUs.*

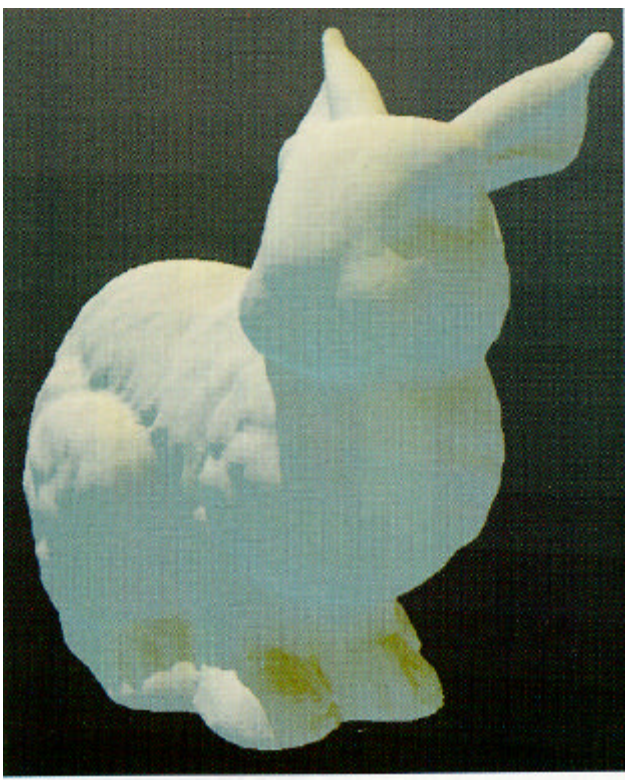

Figure B: *Directionally varying ambient approximation.*

Figure D: *A model with splines, glass, image textures, and procedural solid textures. At 512 by 512 pixels this image is generated at approximately 4 frames per second on 60 CPUs.*