Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models

Michael John Muuss

Leader, Advanced Computer Systems Team
Survivability and Lethality Analysis Directorate
The U.S. Army Research Laboratory
APG, MD 21005-5068 USA

<Mike @ ARL.MIL>

ABSTRACT

Ray-tracing is essential when it is necessary to accurately compute reflection and refraction effects, shadows with penumbrae, and energy transport through participating media. In this new effort, a combination of hardware and software techniques is employed to produce ray-traced renderings of models of enormous complexity (>500,000,000 polygon equivalents) in near-real-time, with the goal being to achieve performance of five television-resolution frames per second. For this class of problem, the ray-tracing technique stands to produce images at a substantially faster rate than any known scanline-rendering system, while simultaneously bestowing all the advantages of ray-traced rendering.

This effort uses a commercial off-the-shelf (COTS) parallel processor array, high-speed HIPPI and ATM networks, and real-time image display hardware. The initial system is being implemented using an 8-node, 96-processor Silicon Graphics Incorporated (SGI) Power Challenge Array. By adding existing BRL-CAD network-distributed ray-tracing software, the result is a system capable of producing real-time ray-traced optical image generation of low-complexity geometry.

The primary goal is to produce real-time imagery for the development and testing of missile automatic target recognition (ATR) systems. This work also addresses one of the pressing needs of the Defense Interactive Simulation (DIS) community by providing the ability to add a physically accurate high-resolution multispectral signature generation node to a distributed simulation when new sensor technology needs to be explored.

May 22, 1995

Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models

Michael John Muuss

Leader, Advanced Computer Systems Team
Survivability and Lethality Analysis Directorate
The U.S. Army Research Laboratory
APG, MD 21005-5068 USA

<Mike @ ARL.MIL>

1. The History of Rendering

Traditionally, the computer graphics rendering of most geometric models has been performed using scanline conversion of polygons or bicubic patches [FOLE84]. Ray-tracing has been considered a ‘‘high-cost’’ alternative to scanline-rendering [KAY79, WHIT80]. Ray-tracing is essential when it is necessary to accurately compute reflection and refraction effects, shadows with penumbrae, and energy transport through participating media [MUUS87a].

If the underlying rendering primitive is the polygon, it can be difficult to render curved surfaces with acceptable accuracy. If an exact surface normal is stored at each vertex of the polygon, then for any pixel on the polygon, an approximate surface normal can be interpolated. This technique is commonly employed in the computer graphics field and gives the surface a fairly reasonable curved ‘‘look’’ to it. However, polygonalization artifacts can still be quite evident along profile edges, and a radar would not be fooled by it.

If the underlying model is a combinatorial solid geometric (CSG) model [MAGI67, JTCG70, JTCG71, MUUS91b] formed from Boolean combinations of primitive shapes, then many curved objects (e.g., spheres, cylinders, cones, tori, etc.) are represented exactly. In general, the CSG representation for a part is much more compact than an equivalent boundary representation, and in the worst case the CSG representation is no larger, as most CSG systems allow boundary representations as one of their ‘‘primitive’’ data types. Thus, to study very intricate models with a high degree of accuracy, there is a double advantage in representing the geometry using CSG techniques and interrogating the geometry using ray-tracing. First, because the representation is more compact, it typically takes less computation to interrogate. At the same time, it is possible to fit larger models into system memory. Second, because the underlying curvature information of the modeled shapes has been retained, profile edges and surface normals are exact, resulting in fewer artifacts in the output.

2. The Task at Hand

In this new effort, a combination of hardware and software techniques are employed to produce ray-traced renderings of models of enormous complexity (50,000,000 CSG solids, approaching 1 billion polygon equivalents) in near-real-time, with the goal being to achieve performance of five NTSC-resolution [CONR80] (720 pixels, 486 active scanlines) frames per second. For this class of problem, the ray-tracing technique stands to produce images at a substantially faster rate than any known scanline-rendering system, while simultaneously bestowing all the advantages of ray-traced rendering.

This effort to produce ray-traced images in real-time exists to support a much larger project [MUUS95a], as depicted in Figure 1. The goal is to accept missile trajectory information from an outboard missile guidance system; use that information to position a virtual missile or aircraft in a complex outdoor *virtual world* filled with realistic terrain, vegetation, man-made structures, and maneuvering vehicles; generate images from that point of view simultaneously in three spectral bands; and deliver those images in real-time to an outboard automatic target recognition (ATR) system [DEIT90a] or man-in-the-loop teleoperation system, which in turn drives an outboard simulation of the missile guidance system, producing the positions and orientations needed as input [DER92].

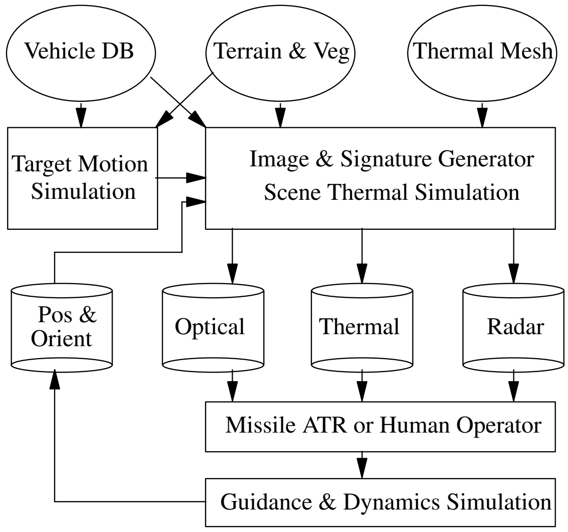

Figure 1. System architecture.

The initial environment is a 12-Km x 16-Km rectangular patch of terrain covering Fort Hunter-Liggett, CA. Elevation data on 1-m grid spacing is used, resulting in 192 million data points (equivalent to 384 million triangular polygons) for the terrain. The location of most real trees has been determined from aerial photography registered to the elevation data, and the geometry of each tree is modeled using approximately 7,000 CSG solids [MUUS95a].

Work on each part of this system is proceeding independently and in parallel, but the most crucial and highest risk component is the real-time image and signature generation component. In our initial tests about 500,000 CSG solids were used to populate a small portion of the terrain with vegetation and a single vehicle. The terrain data were represented as a small number of BRL-CAD *height field* (HF) solids. The equivalent polygon count for this scene is estimated at about 550 million triangular polygons. Current state-of-the-art polygon-rendering hardware, such as the Silicon Graphics RealityEngine(tm) and the Sun/Evans and Sutherland FreedomSeries(tm) machines can only render ‘‘a few’’ million textured and lit polygons per second. If clever field-of-view (FOV) complexity reduction algorithms were employed [TELL91, FUNK93] perhaps along with variable resolution or static *level-of-detail* (LOD) techniques for the vegetation, it might be feasible to reduce the polygon count from 550 million to 50 million. This would still require 50 seconds to render each frame. Furthermore, there are many sensor effects which are difficult to model using polygon-rendering techniques.

Our first initial naive tests were quite encouraging, so the prospects for 5-frames-per-second image generation rates are good. Fortunately, the ray-tracing approach has been aptly dubbed ‘‘embarrassingly parallel,’’ and performance tends to scale linearly with the number of processors [MUUS87d].

3. The Hardware

To achieve a high rate of image production, a very substantial amount of processor power is required. The initial system is being implemented using an 8-node, 96-processor Silicon Graphics Incorporated (SGI) Power Challenge Array, depicted in Figure 2. Each node has 2 Gbytes of 8-way interleaved RAM and 12 MIPS, Inc. R8000 processors running at 75 MHz. Each processor has a dedicated 16 Kbytes of instruction cache and 16 Kbytes of data cache. In addition, there are 4 Mbytes of second-level unified I/D cache. Each of these processors is capable of up to 300 million floating point operations per second (300 MFLOPS), for a system total of over 28 billion floating point operations per second (28 GFLOPS).

The nodes are connected together via 10 Mbit/sec ethernet to a 97th R8000 processor, which we will use for viewpoint control of the simulation. Each node has an 800 Mbit/sec HIPPI (High-Performance

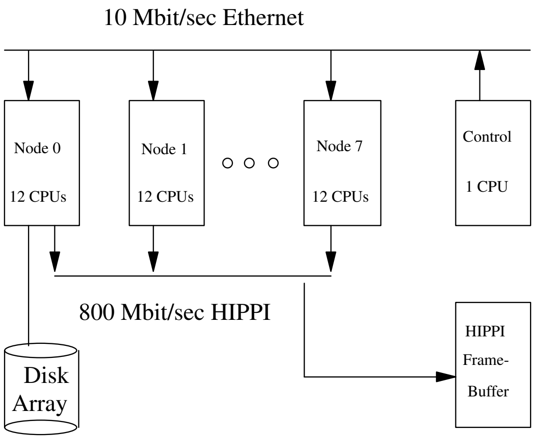

Figure 2. Hardware architecture.

Parallel Interface) to a Network Systems Corporation (NSC) 12-port nonblocking HIPPI cross-bar switch. Each node also has a model VMA-200 interface to a Fore Systems, Inc. Asynchronous Transfer Mode (ATM) network. The link speed is 155 Mbits/sec running over OC3C fiber optics to a dedicated 16-port ATM switch node, providing high-speed network connectivity to ARL's campus ATM network and the DOD's nation-wide DS-3 rate ATM network.†

Node zero is the file server for the Array. It has 15 SCSI controllers and an initial compliment of 48 Gbytes of disk storage. These files are available to the other nodes of the Array via Sun's Network File System (NFS) [WALS85, NOWI89] over the HIPPI interface. In addition, each node has four local disk drives to support the root and /usr filesystems, 8 Gbytes of swap space, and 8 Gbytes of temporary local storage mounted as /var/tmp. A Cray-YMP/EL with dual StorageTek Wolfcreek robot tape silos is attached to both the HIPPI switch and to the ATM switch to provide high-performance NFS access to more than one terabyte (1,000,000 Mbytes) of mass storage.

To permit real-time display of the results, two dedicated PsiTech HFB360 HIPPI framebuffers [PSIT94] are attached to the HIPPI switch. One is configured for 2048-by-2048 pixel display intended for displaying large datasets, and the other is a 1024-by-1024 display. The high bandwidth of the HIPPI framebuffer is essential to realize real-time display of the output. One NTSC frame in 3-byte RGB form at 640-by-480 pixel resolution is just under 1 Mbyte in size (921,600 bytes). To transmit such an image at 30 frames per second requires 27.6 Mbytes/sec of bandwidth. Initial experiments have measured TCP/IP performance over the HIPPI interface from SGI node to SGI node at slightly over 80 Mbytes/sec; raw HIPPI frames which do not require acknowledgment can be transferred at over 98 Mbytes/sec. Therefore there should be no shortage of bandwidth for updating the NTSC resolution images, even given the potential overhead and interference due to coordinating the transmission of eight separate image fragments from the eight nodes of the Array computer.

In addition, each of the eight nodes of the Array will have an SGI Extreme graphics board in it.‡ Each one will have a dedicated monitor, providing another outlet for graphical information. The intention is to use these displays to provide a low-overhead graphical performance monitor.

4. Existing Software and Models

The U. S. Army Ballistic Research Laboratory Computer Aided Design Package (BRL-CAD) is the U. S. Army's third generation solid modeling system [MUUS83, MUUS87a, MUUS91b]. It is Government-owned software written entirely in the **C** programming language [RITC78b, KERN78] and is highly

† Sadly, neither Fore Systems nor Silicon Graphics provides software drivers for the ATM interface under the Irix 6 operating system, so as of this writing the ATM interfaces are nonoperational.

‡ Sadly, due to a bureaucratic error, the original order for the HIPPI framebuffers and the Extreme graphics boards were dropped in favor of an expensive CPU-utilization accounting package. As of this writing, the Array computer does not have a single pixel attached to it by anything faster than an Ethernet, a situation which has substantially hampered developments.

portable, running on more than 12 different hardware platforms — "everything from the Mac 2 to the Cray-2." It is provided in source code form, making it particularly amenable to extension, and already incorporates sophisticated support for shared memory multiprocessors and for fault-tolerant network distributed (non-shared-memory) computation [MUUS87d]. The real-time extensions are being built on top of Release 4.4 of the BRL-CAD Package [MUUS91b].

A collection of CSG models of hundreds of military vehicles, both foreign and domestic, already exists in BRL-CAD form [APPL94]. These geometric models are all Government owned and ready for immediate inclusion into ATR test scenarios [DEIT88]. While polygon-based simulators deem any model over 200 polygons as "high resolution," the BRL-CAD targets are modeled using about 1,000 CSG solids (10,000-20,000 polygon equivalents), and high-resolution BRL-CAD targets range in complexity from 5,000 CSG solids to 40,000 CSG solids (800,000 polygon equivalents).

5. Precursor Experiments

The first software tests were run in December 1994, before the Array computer was delivered, using an existing SGI system similar in configuration but with 18 R8000 processors in one node rather than 12. The system software was IRIX 6.0.0. A beta-test version of the BRL-CAD Package Release 4.4 software was modified to add a pre-release version of **librt/g_hf.c**, the ray intersection module for the HF solid. The goal of this test was to determine whether the BRL-CAD software could reasonably be expected to handle databases of this size, and to get a raw baseline of performance per frame.

The scene was a simple one, including only a single height field solid of one million points (1 Km x 1 Km) and a single target: the **ktank** and three logos. The database comprised a total of 55 solids in 66 regions, running in 0.7 Mbytes of RAM (0.1M initial, 0.2M after gettree, 0.0 for prep, 0.4M during ray-trace). Using 17 CPUs and software semaphores, the fastest frames were rendered in 47 wall-clock seconds. Performance on this test was hampered due to a bug in the vendor-provided parallel processing library **libmpc**, so the semaphore hardware could not be used, and software semaphores had to be used instead, resulting in a substantial CPU penalty roughly estimated to double the total runtime.

Due to some bugs in the HF module, image quality was not acceptable, and a fallback strategy was employed. A 256-m x 256-m patch of terrain taken in 2-m steps was converted into 16,384 pairs of ARB6 primitives (triangular prisms) and substituted for the original HF solid. Now the database comprised a total of 32,822 solids in 66 regions, running in 42.7 Mbytes of RAM (4.8M initial, 26.8M after gettree, 11.1M after prep, 13.1M during ray-trace). The terrain was treated as grey plastic and Phong shaded; no texture map was applied. Using 17 CPUs with software semaphores and rendering at 720 x 486 pixels, the fastest frames were rendered in 871 wall-clock seconds; the slowest frames rendered in 1911 wall-clock seconds. This was horrific performance, but with patience it did succeed in generating image files. Once all the individual frames had been written into disk files, they were transferred to an Abekas A60 D-1 digital video disk, and then recorded onto Betacam-SP master tapes [KENN91]. This resulted in the first video, "*First Quick Height-Field Animation*," which depicted a helicopter flying high over the terrain until it acquired the **ktank** target. While this first effort could not be considered "real-time," it served the important dual purposes of energizing the members of the team, and providing management with visual evidence that the project was worth pursuing, resulting in management commitment to go forward.

The second test used only 6 HF solids of 1 million points each and the same target and flight path. A monochrome texture map taken from aerial photographs sampled on the same 1-m grid as the HF was applied to the terrain solids and was flat shaded. Ray-tracing time was less than 60 wall-clock seconds per frame on a single 150-MHz R4000 Indigo2, with a resolution of 720 x 486 pixels. The BRL-CAD Benchmark has shown that one R8000 CPU is somewhat faster than this workstation, so 96 processors should be able to render this database at better than 1.5 frames per second. These results were recorded as "*K-Tank on Texture Mapped Height Field*." Unlike the first attempt, this test produced quite good looking results at a reasonable rate.

The third test was designed to be more taxing. The scenario was to fly a Tomahawk missile from the western edge of the height field data, following ridges and valleys along a circuitous course until finally screaming straight down the runway of the military airfield and "destroying" the small stand of trees located just beyond the end of the runway. Keyframes for the motion were hand-picked, and crude flight dynamics for the missile were approximated using some custom programs [JOHN95]. Upon viewing the first take, it was clear that improvements in the flight dynamics code were needed, and a second version was computed, resulting in the video “*Tomahawk Missile Flight*.” This sequence was computed using 18 R8000 processors, and achieved an output frame rate better than one frame every 30 clock seconds.

The fourth test was planned specifically for the Array computer. A grove of trees was added, and the monochrome texture values were *stacked* with the Phong shader, greatly improving the look of the terrain and doubling the computational complexity due to having to fire an additional ray per pixel to test for light source (sun) visibility. This expanded the database to 506,618 solids in 3,355 regions running in 1.8 Gbytes of RAM (4.4M initial, 944.1M after gettree, 830.0M after prep, 64.6M during ray-trace). Using 12 CPUs with hardware semaphores and rendering at four times NTSC resolution (1440 x 972 pixels), the fastest frame rendered in 1690 wall-clock seconds. If the resolution were to be dropped back down to 720 x 486 it would have taken 423 wall-clock seconds (7 minutes). If all eight nodes of the array were employed, it would have rendered in 52 seconds per frame.

The fourth test included the grove of trees as well as an articulated missile and sun visibility rays, and on a system with 12 R8000 processors produced an output frame roughly every 7 clock minutes. Clearly the factor of 15 slowdown between the third and fourth tests was not due only to the decreased number of processors (18 to 12) and the enabling of the Phong shader. The extreme size of the database (roughly 500,000 solids) resulted in a working set of 1.8 Gbytes of RAM. It is not yet known how much of this performance loss is due to processor page table management overhead and memory bus contention and how much can be attributed to algorithmic problems associated with such a gargantuan geometry file.

The fifth element of the work completed to date was to add “quick and dirty” first-generation-style forward-looking infrared (FLIR) sensor effects [MOUL95], which incorporated the bidirectional scanning pattern and the AC coupling effect of the sensor and then added a 10-frame left/right/left line-correlated noise sequence in a continuous rolling pattern. This produced images with system noise properties bearing a striking resemblance to what would be observed on the display of a genuine military FLIR. The sensor effects were trivial to apply to the existing image files, taking less than 1 CPU second per frame on an R4000 Indigo workstation. All these results were combined together to produce the video “*Optical and IR Missile Sensor Simulation for ATR: Preliminary Results*.”

6. Assessing the Raw Hardware Potential

The ultimate goal is to render complex scenes in real-time simultaneously in three spectral bands. To get started, we need to begin with a much less ambitious goal: to produce a system capable of real-time ray-traced optical image generation with *low-complexity* geometry. To assess the raw potential of the Array computer, the BRL-CAD benchmark test was run on a single 12-processor node. A 512 x 512 pixel rendering was performed on each of the five benchmark databases, with the results given in Figure 3.

db	solids	walltime	cpu	fps
moss	6	5.36	55.51	1.49
world	8	9.09	104.27	0.88
star	113	7.91	91.35	1.01
bldg391	203	9.57	110.09	0.83
m35	1126	13.06	152.40	0.61

Figure 3. One node performance and estimated Array fps = 8/walltime.

Assuming that the full parallel potential of the Array computer can be utilized without undue losses due to overhead, it is clear that for very simple geometries like “moss,” the Array can already run at a rate of 1.5 frames per second. While the “moss” database has only 6 CSG solids in it, four of those solids are curved (a sphere, an ellipsoid, a torus, and a truncated general cone with elliptical ends of unequal eccentricities), so the equivalent polygonal database would have about 1000 triangles. If a set of software optimizations can be found to improve the existing speed by a factor of 3.5 or better, then our goal of rendering simple geometry at 5 frames per second can be realized.

7. Software Extensions

This remainder of this paper discusses the plans for advancing from the current state of affairs to the goal of 5 frames per second, and beyond. A number of these efforts are loosely coupled and can be tackled in parallel or out of order, depending on the availability of manpower and hardware resources.

7.1. Tight Synchronization

The existing network-distributed ray-tracer **remrt** [MUUS87d] was intended primarily to support two main modes of computation: (1) batch computing of animation sequences of many frames, where the full details of the animation were entirely described by an animation script file, and (2) high-speed "on-demand" rendering of a single frame as quickly as possible. In both cases, relatively small spans of pixels from the frames to be rendered could be allocated across many processors, and spans of results would not arrive back at the **remrt** dispatcher machine "too fast" for the dispatcher to handle. Strict synchronization was never envisioned as a problem for **remrt** as the returned pixels were written directly to the screen regardless of the frame from which they came.

The real-time nature of this new system challenges several of the design assumptions of **remrt**. For one thing, the rate of image production is so high that is not desirable to attempt to change the relationship between processor nodes and screen area *within* a single frame-time, as is done by **remrt**. At best, the work assignments will be changed on a frame-to-frame basis, based upon the assumption of strong frame-to-frame coherence, causing only modest changes in local image complexity and thus processor demand, between any two frames.

Output to the framebuffer must be tightly synchronized, so as not to introduce any temporal artifacts into the display. Secondarily, the output bandwidths required are such that routing all returned pixels through a single processor would be likely to create a bottleneck. Fortunately, the HIPPI framebuffer provides a solution to both of these problems. The HIPPI framebuffers are double-buffered, so that the pixels for the next frame can be built up in the back (nondisplayed) buffer, and then when the proper time has come, the back buffer can be moved into the front (display) buffer during a vertical retrace interval. While the image is being built up in the back buffer, the HIPPI framebuffer can accept intermingled transmissions from multiple nodes on the HIPPI network. Thus, the final image does not need to be assembled in its entirety anywhere except in the actual framebuffer.

A global "master" process is needed, serving much the same purpose as the **remrt** dispatcher process, providing the point-of-view for the next frame to be rendered, as well as indicating when the back buffer of the framebuffer has become available for transmissions of pixels for that frame. This process is tentatively named **rtsync**, and the ray-tracing process running in parallel on each node of the Array is tentatively named **rtnode**. As each **rtnode** completes the rendering task for its portion of that frame, it will send its pixels over the HIPPI network to the back buffer of the framebuffer, and then will notify **rtsync** that the framebuffer has been loaded. **rtsync** will respond by notifying **rtnode** of the next point-of-view and screen region to be rendered, so that rendering can get started immediately. At some later time, **rtsync** will asynchronously notify each of the **rtnode**s that the back buffer of the framebuffer is available and that pixel transmissions are authorized again. This flow of control is depicted in Figure 4. It is unfortunate that the 97th processor does not have a HIPPI connection, as the communication between **rtsync** and the HIPPI framebuffer is based on raw HIPPI packets and not on the InterNet protocols [POST81, DCA83] used on the Ethernet. Either one processor on the Array will be held back to run **rtsync**, or the 97th processor will run **rtsync** and one of the **rtnode** machines will be imposed upon to relay the exchange-buffers command from the Ethernet to the HIPPI framebuffer.

7.2. Viewpoint Control

The BRL-CAD multidevice geometry editor **mged** [MUUS83] is a convenient way to view geometric models and to navigate through complex scenes. It generates wireframe and polygonal displays with user-settable resolution (drawing tolerance). It includes a **libpkg** based interface to an outboard Virtual Reality Manager **vrmgr** which is used to coordinate multiple **mged** instances running on different machines to drive arbitrarily configured multiscreen displays (including stereo and head-mounted displays, 3-, 4-, and 9-screen "caves," etc.) representing virtual "viewing rooms," and then to further coordinate multiple viewing rooms (i.e., multiple independent users) within the same geometric model.

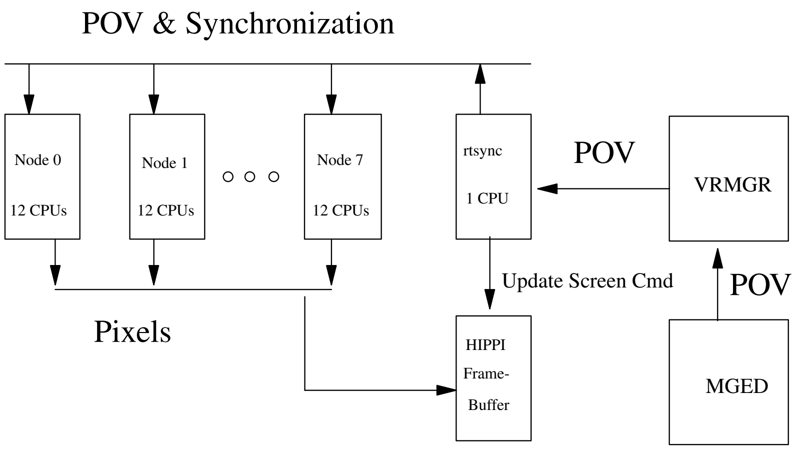

Figure 4. Control flow.

The **vrmgr** uses a succinct protocol to convey point-of-view (POV) changes. **rtsync** will be configured to accept the **vrmgr** protocol, immediately permitting the real-time rendering to be controlled by an **mged** session. This will be a powerful vehicle for testing and demonstrating the system, giving it a familiar user interface and saving the effort of having to code a new user interface. In addition, the **vrmgr** protocol is so trivial that additional applications can be interfaced with the addition of one **sprintf()** and one **pkg_send()** subroutine call.

7.3. Incremental Rendering

If for some reason a ray-tracing node does not complete the rendering of its assigned scanlines within the time allotted, the naive implementation would be forced either to write black pixels in the unrendered area or to reuse the pixels from the previous frame. This could produce bizarre edge discontinuities in the displayed image, as well as a whole host of other bad artifacts.

The main BRL-CAD ray-tracer **rt** includes an option for "incremental mode" rendering. In this mode, rendering is implemented as a process of subdivision. First a single ray is fired. Then, the screen is subdivided into four parts by evenly subdividing in the vertical and horizontal directions. The previously fired ray was the lower left corner; three more rays are fired. This process recurses until full resolution has been reached.

For real-time ray-tracing, this incremental approach offers the possibility for "fail soft" rendering. If the entire screen rectangle assigned to a ray-tracing node is rendered in this incremental manner, then if the rendering is not completed by the end of the assigned time interval, the output can be safely written to the display, and the display will be reasonable and comprehensible. While having different portions of the display rendered at different resolutions is not desirable, the artifacts due to differing resolution levels in the display will certainly be less objectionable than having temporally "out-of-date" pixels on the screen.

If the master display processor detects when the POV is not changing from frame to frame, then increased rendering time can be allotted, permitting a high-resolution image to "fill in." The effect will be not unlike that seen with most workstation-based volume rendering packages.

7.4. HIPPI Framebuffer Support

Two types of software support for the HFB360 [PSIT94] HIPPI framebuffer are required. First, a framebuffer module for **libfb** [MUUS89a] needs to be constructed, so that all the existing BRL-CAD framebuffer applications can access the HIPPI framebuffer display. Starting with a **libfb** module will enable easy testing, as there are only a very small number operations that the framebuffer library supports, and there are existing user programs (commands) to test them. The HFB360 presents a classic set of framebuffer operations as hardware primitives, which map almost directly to **libfb** operations.

The HFB360 exhibits only one oddity: while pixel addressing in the HFB360 is first quadrant, the same as **libfb**, multiple scanline transfers start at the indicated scanline and proceed *down* to lower numbered scanlines. That is, pixels are loaded from left to right and top to bottom. This means that for multiple scanline transfers, either the host CPU is going to have to copy the buffer presented to **libfb** to reorganize the data, or a separate HIPPI transfer will have to be initiated for each scanline. For the 2048-by-2048 pixel HIPPI framebuffer, a scanline is only 6144 bytes long. For the HIPPI bus, this is a relatively short transmission; it will therefore be more efficient to have the CPU reorder the data so that the entire multi-scanline block can be transferred in a single HIPPI transmission. This will require the **libfb** module to maintain a local copy of the pixel data, in the organization native to the HFB360, requiring 12.5 Mbytes of RAM.

The **libfb** support should be carefully constructed in terms of macros and a set of reusable support subroutines which express a software interface to the two dozen or so native HFB360 hardware operations, so that **rtnode** will be able to call those routines directly. By formatting the data in HFB360 native organization to start with, the penalty of an extra data copy can be avoided. Really only three operations will be required: **rtsync** will have to establish the initial operating mode of the HFB360 (probably by just calling the **libfb fb_open**() routine), each **rtnode** will request the direct transfer of an entire rectangular multiscanline pixel block to the back buffer of the framebuffer, and **rtsync** will command the transfer of image data from the back to the front (visible) hardware buffer.

The high speed of the HIPPI bus makes it practical to transmit 30 images a second of 1024 x 1024 x 3 bytes each while only consuming 94 of the 100 Mbytes/sec available bandwidth. It is envisioned that the real-time version of **rt** will always run the framebuffer in 1024 x 1024 mode and perform the calculation in incremental mode, so that when additional processing time is available the full resolution of the display device can be employed. It may also be necessary as an option to operate the framebuffer in a 640 x 480 pixel mode so that the output of the display can be recorded directly onto NTSC compatible videotape. If this is not possible, then some manner of real-time scan-rate converter unit will be required for making videotapes.

7.5. Code Optimization

So far no attempt has been made to optimize the BRL-CAD software for the MIPS R8000 processor, and it is very clear that the R8000 will benefit from such optimization more than general-purpose microprocessors would. The R8000 has six independent functional units, but it is very difficult to keep them all busy. This is partially due to a series of hardware design compromises, and partially due to limitations in the compiler. The **C** compiler for the R8000 has the notion of *software pipelining*, where it unrolls and reorganizes instructions in order to make best use of all the functional units.

There are several coding patterns which are difficult for software pipelining algorithms in the compiler to adequately handle. If the compiler encounters an **if** statement with a subroutine call on either side of the branch, it tends to give up on pipelining the entire block. Unfortunately, the defensive programming style used internal to BRL-CAD's **librt** modules tends to contain code blocks like this:

```
for( afp = &arbp->arb_face[j=arbp->arb_nmfaces-1]; j >= 0; j--, afp-- ) {
	dxbdn = VDOT( afp->peqn, rp->r_pt ) - afp->peqn[3];
	dn = -VDOT( afp->peqn, rp->r_dir );
	if (rt_g.debug & DEBUG_ARB8)
		rt_log("arb: dn=%g dxbdn=%g s=%g0, dn, dxbdn, dxbdn/dn);
	...
}
```

This presents the compiler with a mysterious subroutine call in the middle of a "high-performance" loop. The compiler has no idea that the variable *rt_g.debug* will only be nonzero for debugging, and never during real-time production runs. In order to allow the compiler's software pipelining to optimize this loop, it may be necessary to reorganize the code into two blocks, one with debugging and one without, or maybe even use conditional compilation directives to eliminate the debugging code entirely when compiled for real-time use.

The **for** loop presented previously has another, more subtle barrier to performance. While the **arb_face[]** array is a contiguously allocated block of memory, the way the loop has been written using pointer **afp** and the pointer auto-decrement operator may disguise this fact, preventing the compiler from unrolling the loop or at the least putting next-iteration memory prefetches at the bottom of the loop.

Beyond that, there is a piece of application-specific information that the compiler simply cannot know: the variable **arb_nmfaces** can only take on the values four, five, or six. If this knowledge were made explicit, the code could be unrolled into two conditional instances of the loop's interior block, followed by four unconditional instances, back-to-back. Having such a long stretch of code with no branching would allow the compiler to move memory fetches backwards far enough to overcome the latency of the memory system. While such source code optimizations will bring performance improvements on nearly all architectures, on the R8000 the mismatch between processor rate and memory latency is sufficiently large that a factor of 3-5X speedup is possible.

The majority of **librt** and the whole BRL-CAD programming style were established in the early 1980s according to good University-taught principles, with the underlying assumption that memory access time is uniform across the entire address space, and does not change over time. All data structures are dynamically allocated and are joined to each other using doubly linked lists.

The uniform access time memory model was certainly valid for noncached nonpaged machines like the DEC PDP-11/45 [DEC76]. These assumptions have become less and less valid over time. For example, the DEC VAX-11/780 [DEC82] added the translation lookaside buffer (TLB); for pages of memory not listed in the TLB, several additional round-trips to memory were required to build the necessary TLB entry before the original memory fetch could issue. The R8000 processors being used for this project have a two-level cache. Each processor has a dedicated 16 Kbytes of instruction cache and 16 Kbytes of data cache. In addition, there are 4 Mbytes of second-level unified I/D cache. The cache lines are 128 bits (16 bytes) wide. The upshot of this is that if one 64-bit **double** value is loaded from an address in memory, then the **double** located in the next word is already resident in the first-level cache, where it can be accessed more than ten times faster than if it had to be loaded from primary memory. The same kind of advantage accrues, on a smaller scale, in regards to the TLB; randomly accessing addresses throughout main memory will constantly require new TLB entries to be loaded, at a penalty of more than a dozen memory cycles per entry.

Architectures with strongly hierarchical memory systems like the R8000 perform very well when sequentially accessing every word of an array and very poorly when randomly accessing values from memory. Viewed from the macro scale, **librt** tends to fall somewhere in the middle of that continuum. Overall locality of reference is good, related data structures are block allocated, and the gratuitous (from a performance point of view) software layering so typical of **C++** programs has been avoided. At the same time, the software is highly modular. The most computationally intensive subroutines only perform a few hundred floating point operations before either returning a result or calling a companion subroutine. Furthermore, such subroutines tend to also be data-intensive, in that each value retrieved from memory is used only once or twice.

Given that there is more than a factor of 100 difference in memory access time between words located in the cache and words located in pages not mapped in the TLB, it seems clear that a very modest investment in code tuning can pay off in substantial performance gains.

One of the most difficult to optimize memory reference patterns is generated by a short loop which visits all the structures which are members of a linked list, performing very little computation on the elements of each structure. Since each structure contains a pointer to the start of the next structure, the compiler can generate at best one word of prefetch per iteration. The obvious remedy for this problem is to store a small array of data elements within each member of the linked list, so that at a minimum they will all be colocated in the same memory page and thus will share a TLB entry. Some BRL-CAD data structures are already organized in this manner, such as the **rt_vlist**, but in most cases to date the extra code complexity did not seem warranted. With a little thought given to organization of the data elements, many additional cache hits might be possible.

7.6. Spatial Coherence

One of the most notable sources of **librt**'s high performance is the nonuniform binary space-partitioning tree that it uses as an acceleration technique [MUUS87a], exploiting the spatial coherence of the geometry to make the computational complexity of the ray/model intersection proportional to the geometric complexity of the model in the local vicinity of the ray. This technique tends to outperform the more traditional uniform spatial subdivision [FUJI85] and oct-tree approaches [KAPL85]. Unfortunately traversing the resulting data structure turns out to be performance limited by the access time to main memory. The algorithm takes an X,Y,Z triplet stored in three processor registers. Starting at the root of the tree, a **cutter** structure (containing only four words) is dereferenced. The first word (**cn_axis**) indicates which axis is being tested, selecting which one of the three processor registers is compared against the second word (**cn_point**). If the register is greater than or equal to the second word, then the algorithm uses the pointer found in the third word (**cn_r**), else the algorithm uses the pointer found in the fourth word (**cn_l**), and the loop repeats until the leaf node is found.

While this code does get two "free" memory accesses from the first-level cache, performance is completely memory limited and the loop cannot be pipelined or unrolled. Some preliminary profiling has indicated that for models with large numbers of solids, as much as 65% of the total runtime on the R8000 is spent traversing this data structure. What is worse, since all 12 processors in a node will be constantly traversing this data structure, even if the root of the tree manages to stay in the second-level cache, this will generate a tremendous amount of memory traffic. With 12 processors generating heavy memory traffic 65% of the time, there is a very real possibility that either the system bus or the aggregate memory subsystem bandwidth will become the limiting factor. For the Tomahawk missile sequence, the space-partitioning tree consumed 830 Mbytes of RAM. For any one frame, the algorithm has a very small footprint inside the tree, but it is still quite likely that both the cache and TLB are hard-hit.

Simply eliminating the space-partitioning algorithm is not beneficial, as that results in a factor of 100 or worse performance loss. Clearly what is needed is another algorithm which exhibits equal or better theoretical performance, while simultaneously being less memory intensive. Fortunately, a candidate algorithm is known: NUGrid [GIGAN90]. NUGrid replaces the binary tree data structure which leads to the leaf nodes with a nonuniformly spaced three-dimensional array of leaf nodes. The index for the proper leaf node can be computed in a few dozen instructions. For small models the NUGrid algorithm may not provide any performance advantage, but for large models running on the R8000 processor it may come close to doubling the overall ray-tracing performance.

In order to further optimize performance, several strategies will have to be employed in parallel: (1) careful run-time profiling of the R8000 processor, executing both the benchmark models and the full-scale models, to determine which portions of **librt** will benefit most from being tuned, (2) examination of the assembly language generated by the compiler, to get a sense for which specific **C** language constructs inhibit good performance, (3) re-examination of data structure access patterns, and (4) consideration of alternative data organizations and data-traversal algorithms which are cognizant of the latency variations between the different levels of a hierarchical (cache) memory system.

7.7. Spectral Coherence

By uniting the three different signature prediction codes into one multispectral prediction code, a substantial efficiency gain is possible. For that area where the FOV of the three sensors overlaps, each main "eye visibility" ray fired from the sensor location into the scene can be reused for all spectral bands. Once the ray strikes an object, additional rays may be needed to compute wavelength-dependent effects, including reflection and reflection. Similarly, one "sun visibility" ray can be used both for shadow generation in the optical model, and in the solar loading calculation for the thermal model.

In practical terms, what this means is that the thermal signature can be generated almost for free. By using the same set of rays that the optical rendering used and passing them to a set of thermal *shader* routines instead of the conventional optical shader routines, the scene radiance can be computed. Each **rtnode** can apply the FLIR sensor effects to the block of thermal pixels just before sending them to the HIPPI framebuffer. The filter kernels of the sensor effects algorithm are well suited for the R8000 processor architecture and should add a bare minimum of extra runtime. Thus, 512-by-512 pixel optical and thermal images could be displayed side-by-side in the 1024-by-1024 pixel framebuffer at almost no additional cost.

Generating the pseudo-radar signature [MUUS95a] will require additional rays to be fired, causing the frame rate to be cut in half. Only initial rays which did not intersect the terrain or the sky need to be processed further. Even if the POV is near the ground and close to the target, fewer that 25% of the initial rays are likely to need additional processing, and most of them will probably not strike any metal. Only rays which strike metal objects (vehicles, man-made structures, iron-rich boulders) need to generate reflected rays and recurse.

The **rtnode** ray-tracer will use the proven self-scheduling automatic load-balancing pixel-to-processor assignment algorithm used in **rt** and **rtsrv**. Because of this, an irregular distribution of metal objects in the scene will not cause a workload imbalance among the 12 processors of one node, although it may cause some nodes to take longer to complete than others, based upon the metal content of that segment of the scene.

7.8. Temporal Coherence

Even when flying at a speed of 220 m/sec over the terrain, there is a substantial amount of coherence from frame-to-frame for objects in the distance. One of the natural byproducts of ray-traced scene generation is a *range map* for each pixel. Similarly, it is possible to determine the *optical flow map* of all pixels, in effect producing a map which shows how (for the far-field) the most recent change in view orientation relates pixels in the previous frame to pixels in the current (unrendered) frame. In practice, the flow map would be implemented as a procedure rather than as an tabular map.

For example, if a sharp turn to the left occurs between two frames, the pixels on the left side of the previous frame will show up closer towards the center of the FOV, and the pixels from the center will show up towards the right side of the new frame. Using these two elements, a simple algorithm to exploit temporal coherence would proceed like this:

(1) If the pixel to be rendered is close to the center of the screen, then render this pixel normally.

(2) If the flow map indicates that this pixel points to something that was not in view in the previous frame, then render this pixel normally.

(3) If the mapped-to pixel from the previous frame has a range map entry which says that pixel is not in the far-field, then render this pixel normally.

(4) The pixel is unlikely to have changed substantially, record the mapped-to pixel's spectral and range map values as the value for this pixel.

Given that roughly 60% of most outdoor scenes is sky and distant horizon, even with a fast-traveling viewpoint it should be possible to avoid rendering 25% of the pixels in the scene. Of course, such an approach would introduce artifacts into the rendered image when a foreground object such as a tree shifted to obscure a formerly visible horizon pixel. For applications such as ATR algorithm testing where image accuracy and quality are paramount, this feature would have to be disabled, but for applications where a human observer was performing a noncritical task, the improvement in frame rate ought to be worth the presence of artifacts around the edge of the screen.

Stochastic techniques can be employed to further reduce the possibility of artifacts due to obscuration. The algorithm only needs to detect the change from far-field to near-field status of a pixel; pixels changing the other way will be rendered normally on this frame. Randomly select 5% of all the pixels which could be retrieved through the flow map for direct rendering. If the direct rendering produces a range value which is not in the far-field like it should have been, then tag the three neighboring pixels so that they will not be flow-mapped either, rerendering any which have already been completed. Thus, which three of the eight possible neighbors should be marked is determined from the gradient of the optical flow.

8. Summary

Real-time ray-tracing is on the verge of becoming a reality, at least for a special class of models on an expensive ($6 million) piece of hardware. Thanks to the miracle of VLSI, within 5 years, this level of processing power will be found on large ‘‘departmental-sized’’ super-workstations at one tenth the cost of the current system. This research is developing general purpose algorithms and free software to take full advantage of such systems when they become more widely available.

In the meantime, in addition to supporting the primary goal of producing real-time imagery for the development and testing of missile automatic target recognition (ATR) systems [MUUS95a], this work addresses one of the pressing needs of the Defense Interactive Simulation (DIS) community, by providing the ability to add a physically accurate high-resolution multispectral signature generation node to a distributed simulation where new sensor technology is being explored. Considering the high cost of test-firing million-dollar missiles, this research should pay for itself many times over in the coming years.

9. Acknowledgments

The author would like to thank Dr. Paul Deitz for providing his unflagging support and encouragement and for providing an environment that makes good research possible. The author would also like to thank Chris Johnson for his early enthusiasm for ray-tracing height fields and flying Tomahawk missiles; without him, this project would not have gotten off the ground.

Additionally, the author would also like to thank Phil Dykstra, Paul Stay, and Chuck Kennedy for their efforts over a 2-year period to obtain the Array computer to support exactly this kind of DOD project. And finally the author would like to thank his supervisor, Lex Morrissey, for his patience and understanding as this project has slowly come to life.

This work was supported in part by a grant of HPC time from the DOD HPC Shared Resource Center at the U.S. Army Research Laboratory on the Silicon Graphics, Inc. PowerChallenge Array computer.

References

APPL94.
K. A. Applin, *A Catalog of BRL-CAD Target Descriptions,* ARL-MR-145, U. S. Army Research Laboratory, APG, MD, August 1994.

CONR80.
Conrac, *Raster Graphics Handbook,* Van Nostrand Reinhold, New York, 1980.

DCA83.
DCA, *MIL-STD-1777: Internet Protocol,* Defense Communications Agency, U. S. Government Printing Office, 12 August 1983.

DEC76.
DEC, *Digital PDP-11/70 Processor Handbook,* Digital Equipment Corporation, Maynard, MA, 1976.

DEC82.
DEC, *VAX Technical Summary,* Digital Equipment Corporation, Maynard, MA, 1982.

DEIT88.
P. Deitz, W. Mermagen, Jr., and P. Stay, "An Integrated Environment for Army, Navy, and Air Force Target Description Support," *Proceedings of the Tenth Annual Symposium on Survivability/Vulnerability*, April 1988.

DEIT90a.
P. H. Deitz, M. J. Muuss, and E. O. Davisson, "Issues in Automatic Object Recognition: Linking Geometry/Material Data to Predictive Signature Codes," *First Proceedings of the Society of PhotoOptical Instrumentation Engineers (SPIE) Advanced Institute Program on Automatic Object Recognition*, Coco Beach, FL, 21-23 April 1990.

DER92.
S. Der, G. Dome, J. Horger, and M. Lorenzo, *A Multi-Sensor Digital Terrain Board for Processor Test and Evaluation,* Proceedings of IEEE MILCOM Conference, San Diego, CA, October 1992.

FOLE84.
J. D. Foley and A. VanDam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesly, 1984.

FUJI85.
A. Fujimoto, et.al., "Accelerated Ray Tracing," *Proceedings of CG Tokyo '85*, Tokyo, April 1985.

FUNK93.
T. A. Funkhouser and C. H. Sequin, *Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments,* pp. 247-254, Proceedings of SIGGRAPH 93, Anaheim, CA, August 1993.

GIGAN90.
M. Gigante, "Accelerated Ray Tracing using Non-Uniform Grids," *Proceedings of Ausgraph '90*, pp. 157-163, Melbourne, Australia, September 1990.

JOHN95.
C. T. Johnson, *A Low Level Fly-By: Look Mom, No C,* BRL-CAD Symposium '95, Aberdeen Proving Ground, MD, 5-9 June 1995.

JTCG70.
JTCG, *MAGIC Computer Simulation, Vol. 1, User Manual,* Joint Technical Coordinating Group for Munitions Effectiveness, 61JTCG/ME-71-7-1, July 1970.

JTCG71.
JTCG, *MAGIC Computer Simulation, Vol. 2, Analyst Manual,* Joint Technical Coordinating Group for Munitions Effectiveness, 61JTCG/ME-71-7-2-1, May 1971.

KAPL85.
M. R. Kaplan, "Space-Tracing, a Constant Time Ray-Tracer," *State of the Art in Image Synthesis*, Siggraph '85 Tutorial, San Francisco, CA, 22-26 July 1985.

KAY79.
D. S. Kay, *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images,* Cornell Univ, Jan 1979.

KENN91.
C. M. Kennedy, *Video Hardware for Making Movies,* BRL-CAD Symposium '91, Aberdeen Proving Ground, MD, 7-9 May 1991.

KERN78.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Bell Laboratories, Murray Hill, NJ, 1978.

MAGI67.
MAGI, *A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles,* MAGI Report 6701, AD847576, August 1967.

MOUL95.
R. Moulton, Jr., *Sensor Effects,* NVESD Internal Communication, 29 March 1995.

MUUS83.
M. J. Muuss, K. A. Applin, R. J. Suckling, G. S. Moss, E. P. Weaver, and C. Stanley, *GED: An Interactive Solid Modeling System for Vulnerability Analysis,* ARBRL-TR-02480, NTIS AD No. A-126-657, U. S. Army Ballistic Research Laboratory, APG, MD, March 1983.

MUUS87a.
M. J. Muuss, "Understanding the Preparation and Analysis of Solid Models," in *Techniques for Computer Graphics*, ed. D. F. Rogers and R. A. Earnshaw, Springer-Verlag, New York, 1987.

MUUS87d.
M. J. Muuss, "RT and REMRT - Shared Memory Parallel and Network Distributed Ray-Tracing Programs," *USENIX: Proceeding of the Fourth Computer Graphics Workshop*, Oct 1987.

MUUS89a.
M. J. Muuss, *Frame Buffers and Window Systems: Internals of the Silicon Graphics 4D Framebuffer Support for LIBFB,* pp. 54-72, BRL-CAD Symposium '89, Aberdeen Proving Ground, MD, 24-25 October 1989.

MUUS91b.
M. J. Muuss, P. Dykstra, P. Deitz, K. Applin, and P. Stay, *The Ballistic Research Laboratory CAD Package, Release 4.0,* BRL Internal Publication, U. S. Army Ballistic Research Laboratory, APG, MD, December 1991. Volume I: The BRL-CAD Philosophy, Volume II: The BRL-CAD User's Manual, Volume III: The BRL-CAD Applications, Volume IV: The MGED User's Manual, Volume V: The BRL-CAD Analyst's Manual.

MUUS95a.
M. J. Muuss and M. Lorenzo, *High-Resolution Interactive Multi-Spectral Missile Sensor Simulation for ATR and DIS,* BRL-CAD Symposium '95, Aberdeen Proving Ground, MD, 5-9 June 1995.

NOWI89.
B. Nowicki, *NFS: Network File System Protocol Specification, RFC1094,* DDN Network Information Center, Menlo Park, CA, March 1989.

POST81.
J. Postel, "Internet Protocol Specification, RFC-791," in *DDN Protocol Handbook, Vol 2, NIC 50004*, DDN Network Information Center, Menlo Park, CA, September 1981.

PSIT94.
PsiTech, Inc, *HFB360 User's Manual,* Document 956320-001, Revision 3.3, Fountain Valley, CA, June 1994.

RITC78b.
D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1991-2019, 1978.

TELL91.
S. J. Teller and C. H. Sequin, "Visibility Preprocessing for Interactive Walkthroughs," *Computer Graphics*, vol. 25, no. 4, pp. 61-69, Proceedings of SIGGRAPH 91, Las Vegas, NV, July 1991.

WALS85.
D. Walsh, B. Lyon, and G. Sager, *Overview of the Sun Network File System,* pp. 117-124, USENIX: Winter Conference Proceedings, Dallas, TX, January 1985.

WHIT80.
J. T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343-349, June 1980.