Ray Tracing News

"Light Makes Right"

February 2.01, 1994
Volume 7, Number 2

Compiled by:

Eric Haines <erich@eye.com>

3D/Eye Inc

1050 Craft Road

Ithaca, NY 14850

Contents:

Introduction

In celebration of James Joyce's birthday, (or is it Bloomsday, or both? I forget...) we're having a special two-for-one sale. This issue is dedicated to research issues and programmer resources. It includes articles on a variety of rendering related topics, research article summaries, new books, etc. The other issue (v7n1) is more for renderer users. There was enough accumulated stuff that I decided to try splitting things along these lines. Let me know if the split was worthwhile.

There are some worthwhile resources and articles in this issue. Nicholas Wilt's ray tracer in C++ is now available on the net. P.H. Andersen's progress report discusses an acceleration technology I had not heard of before. Erik Jansen reports on some surprising results with acceleration techniques he's exploring. Olin Lathrop takes a position ("Gamma correction is largely a crock") which is both polemical and practical. There are also extracts from the net on parallelism which I found interesting, along with some article summaries, new book listings, CFP's and whatnot.

Some things worth watching for (but haven't been released yet) are "Graphics Gems IV" edited by Paul Heckbert, Academic Press; Sillion & Puech's radiosity book from Morgan Kaufmann; and Ian Ashdown's applied radiosity book. Further on (by SIGGRAPH, I assume) will be Glassner's digital image synthesis book from Morgan Kaufmann, the first full-length book dedicated to computer graphics related theory. If you know of anything I've missed, let me know.

Ray Tracing Roundup

Updated Ray Tracing and Radiosity Bibliographies

The free ray tracing and radiosity bibliographies which I maintain have been updated with references from 1993. These are at `princeton.edu` in the directory `/pub/Graphics/Papers.` The files are `RayBib.10.93.Z` and `RadBib.10.93.Z`

(Eric Haines)

References to Blobby Models, Metaballs, etc.

The following is a bibliography on particle systems compiled by David Breen at Rensselaer, myself, and Bill Gates at UBC. Blobby objects and metaballs are one type of particle systems. This bibliography has been printed as part of the SIGGRAPH'92 course #16 notes: "Particle System Modeling, Animation, and Physically Based Techniques".

(Dave Tonnesen, `davet@dgp.toronto.edu`)

[*I do not include the bibliography here because it is (surprisingly) long. If you want it, you can ask me for a copy. -EAH*]

A Note on SPD Platform/Compiler Results (SPARC II), by David Hook (`dgh@ecr.mu.oz.au`)

With the tweaked benchmarks in RTNv6n3 we removed the backing polygon altogether from the scene - we did this because we were interested in measuring the effect of having such a large object in the scene.

***OORT* - Object Oriented Ray Tracer,** by Nicholas Wilt [and Eric Haines]

`(75210.2455@compuserve.com)`

[I have also woven in my own comments about the book and code into this announcement. Nick is nice enough to make the code available for free on the net, as the disk does not come with the book and normally costs $33 more!]

[*The code for Nick's book is now available at the following sites as `oort10.zip` and `oort10.tar.Z`:*

`wuarchive.wustl.edu:/graphic/graphics/ray`

`ftp.funet.fi:pub/graphics/packages/ray-tracing/oort`

`gondwana.ecr.mu.oz.au (128.250.70.62):/pub`

-EAH]

Extracted from the README:

This distribution contains the source code that accompanies my book, **Object-Oriented Ray Tracing in C++**. If you are interested in ray tracing, or if you want to understand *OORT* in more detail, you can call John Wiley & Sons at 1-800-CALLWILEY (1-800-225-5945) to order it. Alternatively, you can find it in a local bookstore or have them order it for you (ISBN 0471 304 158, $36.95).

[*I proof-read much of this book. It's good for the serious novice programmer wanting to do a ray tracer (the C++ code also seems nice, but I'm just a beginner here); though it's a trade paperback, it's well referenced (for a switch). It follows mainstream professional practices instead of amateur inventions. The layout and illustrations are nothing to write home about, but suffice. The first two thirds of the book walks through the topics below, the last third is a reference manual for the OORT class library and appendices for an intro to C++, GIF display, scene examples, file formats, etc. The code per page amount is around 65% code for the first two thirds of the book, which is pretty high as these things go but is pretty reasonable for a book presenting C++ for novices. Also, the code is well commented, so it's usually not just pages and pages of tokens. -EAH*]

The book assumes some knowledge of 3D computer graphics. It includes lots of references so readers can pursue their own research, too. The topics covered include the following:

- Vectors and matrices for computer graphics.

- Standard Hall shading model with ambient, diffuse, specular, reflective and transparent components.

- Rendering various primitives (planes, rings, polygons, spheres, bounding boxes, quadrics, CSG and algebraic surfaces).

- Ray tracing acceleration. *OORT* implements Goldsmith and Salmon's automatic bounding volume hierarchy generation for speed. It also includes a hand-coded 80x87 assembler routine to intersect a ray with a bounding box.

- Texture mapping. Procedural and solid texturing is described. *OORT* implements a bunch of inverse mappings (sphere, cone, cylinder, quadrilateral, circle).

- Distribution ray tracing for antialiasing, soft shadows, and imperfectly reflective and translucent objects.

- Statistical optimization to make the distribution ray tracing go faster.

OORT is a class library for ray tracing. As such, it does not implement an input language or interactive user interface. Currently, images are described to the ray tracer using calls into the class library. This is surprisingly painless; unlike C code, C++ is almost as intuitive as the input language of a ray tracer. In addition, C++ is much more powerful than any ray tracer input language; it has loop constructs, functions and support for abstract data types. Finally, C++ debugging tools are much more sophisticated than the debugging tools that are available for proprietary ray tracer input languages.

All these protestations are not to say that no interactive user interface for *OORT* is in the works. I plan to develop an interactive user interface for *OORT*, and may provide parsers for other ray tracers' input languages. But for now, if you want to make pictures using *OORT*, you have to develop in C++.

[*If you want to make pretty pictures, get POV, Polyray, Rayshade, etc. If you want to look at some nice C++ code for a vector & matrix library, etc, check this code out. Note that the code does use templates (which some C++ compilers don't have yet, Microsoft's included). Also, it's interesting to note that this code is easier to integrate into existing software packages since it is procedure driven (unlike most free ray tracers out there which are input language driven). Finally, this book is worthwhile as an example of a serious C++ application. Personally, learning C++ by examining the trivial examples in most books is useful for understanding syntax, but grokking the true power of OO is easier when examining a full-blown ray tracer built in C++. There are times when a pure object orientation is not used in this code for simplicity (e.g. bounding boxes are treated differently), but otherwise it's pleasantly polymorphic. -EAH*]

Nicholas Wilt writes:

There are two versions of *OORT*. The June 1993 version was cut for the book. The only people who can get their hands on the June build are those who buy the book/disk set from Wiley.

The freeware distribution was cut in late 1993 (November or December). The differences between the book distribution and the freeware distribution are minimal:

1) The book distribution contains a shareware GIF viewer, but the freeware distribution does not (I figure anyone who can get their hands on the freeware distribution has access to a GIF viewer of their own).

2) The freeware distribution has a few fixes to make *OORT* compatible with Borland C++ 4.0. The number of lines of code affected can be counted on one hand, but the source code won't compile as is for BC++ 4.0.

The "official," current version of *OORT* is the freeware distribution (again, the differences are minimal).

Nicholas Wilt also writes about what he's up to now:

I've been working on the C++ code lately, though not on the rendering engine. I want an object-oriented framework for 3D rendering that decouples the rendering technique from the modelling and quantization and display. That led me to develop an interactive modelling and display program for Microsoft Windows and Windows NT (I got tired of writing C++ source code to render images). You professionals won't be too excited, but there are lots of ray tracing wienies here on CompuServe that might be :-).

Windows is a good environment for this stuff, because it has dynamic linking built in. If I do the renderer right, it should be very configurable and extensible. NT has symmetric multiprocessing and a multitude of networking models built in (RPC, named pipes, sockets, to name a few), so doing a parallel ray tracer should be a snap.

I got tired of waiting for ray tracings to finish, so I did a Z-buffering polygonal renderer.

Partial Evaluation Applied to Ray Tracing, by P.H. Andersen (`txix@diku.dk`)

I have implemented a simple ray tracer in C, and optimized it by partial evaluation to run twice as fast. The unoptimized program is faster than Rayshade 4.0.6 measured on raw intersection speed.

I am currently working on optimizing Rayshade by partial evaluation.

Below is a short description of partial evaluation and the application to ray tracing.

Partial Evaluation

Partial evaluation is a general and automatic program transformation technique that can specialize programs with respect some of their input data.

A partial evaluator is a program which, when given a program and some of its input data (the static data), produces a specialized program. Running the specialized program on the remaining input data (the dynamic data) yields the same results as running the original program on all of its input data. A partial evaluator unrolls loops, unfolds function calls, precomputes expressions that only depend on static data, and reduces expressions depending on dynamic data. The benefit of partial evaluation is speed of execution: the specialized program is often significantly faster than the general program. Refer to [5] for a general introduction to partial evaluation.

For this project we use a partial evaluator for C called C-Mix [1], [2], [3].

Ray Tracing

The usual implementation is by a general algorithm which, given a scene and a ray, performs computation to follow its path. The main loop of the ray tracing algorithm could look like this:

```
for each pixel (x,y) in the picture do
   ray = <compute the primary ray using (x,y)>;
   color = trace(scene, ray);
   plot(x, y, color);
```

Since trace calls itself recursively to model reflected and refracted light, the algorithm is rather time consuming.

Partial Evaluation Applied to Ray Tracing

In all calls to the function trace the scene is the same, which makes partial evaluation highly relevant. Thus we specify that scene is static and ray is dynamic. Given the program and the static scene data the partial evaluator will produce a specialized program, where the main loop will look like this:

```
for each pixel (x,y) in the picture do
   ray = <compute primary ray using (x,y)>;
   color = trace_scene(ray);
   plot(x, y, color);
```

The function trace_scene is a version of trace, which is specialized with respect to the scene. In other words: trace_scene is a new function, which is only good for tracing rays in this one scene, but which is often faster than the general algorithm.

If the picture generated is big enough, the time to specialize plus the time to run the specialized program will be less than the time to run the original program (often true, since we all know how time consuming ray tracing can be!). Further, in situations where the specialized program is run more than one time, the gain is much greater of course. This is especially the case for animations where only the viewpoint changes and/or only part of the scene changes. Another example is interactive design where one wants to try out different surface data, light sources, or eye positions.

Related Work

Mogensen specialized a very modular ray tracer written in a functional language [6], showing that the administrative overhead can be removed by partial evaluation.

Hanrahan has a 'surface compiler' which accepts as input the equation of a surface and outputs the intersection code as a series of C statements [4]. He reports speedup of 20 %.

References

[1] Andersen, L.O. Partial Evaluation of C and Automatic Compiler Generation (Extend Abstract). Compiler Construction, Paderborn, Germany. October 1992. (Lecture Notes in Computer Science, vol. 641).

[2] Andersen, L.O. Self-Applicable C Program Specialization. Proc. of ACM Symposium on Partial Evaluation and Sematics-Based Program Manipulation, PEPM'92. 1992.

[3] Andersen, L.O. Binding-Time Analysis and the Taming of C Pointers. Proc. of ACM Symposium on Partial Evaluation and Sematics-Based Program Manipulation, PEPM'93. 1993.

[4] Hanrahan, Pat. Ray Tracing Algebraic Surfaces. Computer Graphics, Volume 17, Number 3. July 1983.

[5] Jones, N.D., Gomard, C.K, and Sestoft, P. Partial Evaluation and Automatic Program Generation. Prentice-Hall. 1993.

[6] Mogensen, Torben. The application of Partial Evaluation to Ray-Tracing. Master Thesis, University of Copenhagen, Dept. of Computer Science, DIKU. 1986.

Comparison of Ray Traversal Methods, by Erik Jansen
`(fwj@duticg.twi.tudelft.nl)`

In RT-News Vol 5, nr.1, July 1992, we reported a comparison of the DDA-grid traversal as implemented (standard) in RayShade with a recursive bintree (BSP) traversal implemented in RayShade by Wim de Leeuw. Although the BSP-method in all cases performs reasonable well (without additional tweaking), the grid method wins when some tweaking is allowed (leaving the groundplane out of the grid, etc.).

I disagree with those authors that take the position that a method should be fully automatic (without adjusting parameters by hand). Of course, in practice it would be handy to have a fully automatic method, but personally I think that we have two different problems: (1) which method is fastest (with the best settings possible) and (2) how can we automate the method to find the optimum setting of parameters algorithmically. In addition, I think that we should be liberal in allowing having a two-level grid, i.e. a main grid with in certain partitions or macro-objects a subgrid. So, in my opinion, a grid method could be hierarchical as well (and still differ basically from the adaptive methods as bintree and octree). With these assumptions the grid method is (still) the winner.

We have continued our experiments with the implementation of a DDA-bintree traversal in RayShade. This was done by another student, Pim van der Wal. The method uses the standard RayShade DDA-traversal that traverses a (two-level) 3D virtual grid that is mapped to the bintree subdivision with an (hierarchical) spatial index in the style of the EXCELL method of Tamminen (the alternative of a hash table needs less memory but is slower). From a first implementation we learned that the timings for the SPD models did not really differ from those with the BSP-method, so we may conclude that the recursive traversal is indeed an efficient traversal for the bintree, and we did not persue the DDA-bintree approach any further.

With respect to comparing the grid method with the bintree (or octree) methods, we were interested to know whether the three methods would behave differently when the density of the model data was increased locally. Intuitively we thought that this would benefit the adaptive methods and harm the grid method. Therefore we made different linear approximations of the teapot model, increasing the amount of polygons from 10,000 to 60,000. For the subdivision of the bintree a subdivision criterion of max. 20 polygons per cell was used. For the grid a resolution of nxnxn was chosen where n = (#polygons)^1/3 (eg. 22x22x22 for 10712 polygons).

polygons	grid	bsp	dda-bintree
10712	164.7	197.3	229.9
20592	206.0	216.6	250.7
30800	207.8	238.1	279.2
43056	232.8	258.8	322.4
53592	246.3	270.7	328.8
61256	267.0	291.9	345.7

If you plot these figures, you see three parallel lines!

[*This is a pretty surprising result, if you think about it - increasing the number of polygons affects all schemes in the same fashion, on the face of it. Erik will be making more results and ideas available from time to time, since this is work in progress. -EAH*]

I wrote Erik in response:

One area I wish was explored more formally is characterizing the complexity of a model with regards to its rendering method. One interesting scheme Cornell tried a decade ago (in the Weghorst & Hooper & Greenberg TOG paper, I think) was to do things like zoom way out (so that the model was a speck on the screen), do a normal rendering, and zoom way in (so a tiny portion of one surface was rendered. A graph of time spent vs. field of view angle might be interesting in comparing schemes.

Varying the size of the background plane polygon and charting the effect on various schemes might also be illuminating. You and I can guess what the results probably will be. But you don't know until you try, as you point out with your recent experiment. It would also be interesting to me to try to characterize the models themselves, e.g. graph the number of objects per cell for a given set of models and see if there are any correlations between these graphs and relative performance of schemes (e.g. we suspect "when there are a large number of grid cells with a large number of items, the octree scheme performs better than DDA").

Anyway, these are just some experiments that your research made me think about. What would be useful is some more knowledge of statistical methods, as I'm not sure how you could characterize such graphs into a few variables, for example, and then see the correlation between these variables and performance times. Such research could lead to some heuristics that would be quite useful: "when the graph looks like this, use BSP, else use DDA".

Sphere Tessellation//Gamma Correction, by Olin Lathrop

`(olin%cognivis@uunet.uu.net)`

[*These are comments by Olin on RTNv6n3. Olin is the president of Cognivision, a visualization company, and has been in rendering for more than a decade. -EAH*]

Simple Sphere Tessellation

The method as described [make three subtriangle from one triangle, by dividing from the centroid] would end up with long and thin triangles near the edges of the original triangles. Ideally you want a method that is completely symmetrical in recursion. That doesn't really exist, but you can get much better results by subdividing each triangle into four triangles by using the midpoints of the edges. I tried this about ten years ago with quite reasonable results.

The problem is that the four newly created triangles aren't exactly the same size due to the sphere's curvature. The center one is always larger than the other three. This effect falls off rapidly as the original triangle gets smaller. The effect is not that strong, but is visually quite noticeable if you facet-shade the sphere, especially when starting with a tetrahedron (the worst case). Starting with an octahedron as Mike Castle suggested is already noticeably better.

You can apply the same technique to subdividing a quadrilateral into four smaller ones. In this case, start with a cube.

Gamma Correction

Gamma correction is largely a crock because the fundamental assumption is wrong. Yes, there is a (reasonably) reliable relationship between voltage and phosphor emission power that the standard "gamma" formula models well enough for our purposes. Unfortunately, this is not at all what the relationship from pixel intensity value to visible brightness actually looks like. Major reasons why not include the monitor offset (black level) control, and the CRT's reflection of ambient light. The amplifiers can also be quite
non-linear at low voltage levels.

All this amounts to black not really being black, which throws a big wrench into the gamma formula. Here is how I think of and deal with the problem:

The problem really comes down to mapping an image with a large dynamic range (256:1) to an output medium with considerably less. Ektachrome slides can get to 150:1, glossy prints with careful illumination 80:1, matte prints maybe 40:1, newspapers 15:1. A well adjusted monitor in a dark room maybe 100:1 if everything is perfect. Figure 40:1 (if that much) for more normal monitor situations. Contrast this to real world scenes in a office, which can easily reach 500:1.

Clearly, something needs to get squashed in the process displaying or making a hard copy of most images. The quest is to squash while maintaining the highest perceived "likeness" to the true image. Humans perceive intensities logarithmically, not linearly. Since we want to do image manipulations in perceptual linear space, this means manipulating brightnesses in mathematical log space. Humans also tend to "tune out" details in dark areas of a scene. There was a lot of interesting research on this stuff done by Kodak back in the 1930s.

Between the black level problem, the way our perception system works, and a host of unpredictable environmental variables, it seems pointless to try to come up with a formula that can then be "corrected" for. Even if you could come close, my points are:

1) Such a formula couldn’t be reasonably approximated by anything as simple as one "gamma" exponential.

2) Even if the function was known, you don’t want to "correct" for the best linear mapping anyway. You want the best logarithmic mapping.

So, here’s what I do. My software can map the input black to white range to any other range, and can independently apply a "brightness" factor. The brightness factor is exponential in nature, but doesn’t effect full black or full white. A value of 0 leaves everything alone. Positive values increasingly slosh intensities smoothly to the white end, while negative values slosh towards black. This doesn’t try to model what the hardware is actually doing, but provides enough controls to make things look good.

I have used this technique to pre-treat images for film recorders, printers, and other output media with great results.

Computational Geometry On-Line Bibliography, by Bill Jones (`jones@skdad.usask.ca`)

The computational geometry community maintains its own bibliography of publications in or closely related to that subject. Every four months, additions and corrections are solicited from users, after which the database is updated and released anew. As of September 1993, it contained 5356 bibtex entries. It can be retrieved via anonymous ftp from `cs.usask.ca` in the directory `pub/geometry`. The file `geombib.tar.Z` therein is the bibliography proper, along with supporting software and documentation; `geom.ps.Z` is a Postscript formatting of the database; and `o-cgc19.ps.Z` is an overview which was published during 1993 in SIGACT News and the International J. Comput. Geom. Appl. The file `ftp-hints` provides more detailed retrieval information.

Summary of Advanced Rendering Papers from Eurographics '93, by Erik Jansen (`fwj@duticg.twi.tudelft.nl`)

[*The exact references can be found at* `princeton.edu` *in* `/pub/Graphics/papers` `RTBib` *and* `RadBib` - *EAH*]

Takamura, et al.: a skylight illumination model based on scattering due to air molecules and aerosols. Comparison of two methods to calculate the sky illuminance received by a point: a band source method and a parallelepiped (a sort of hemi-cube) method.

Blasi, et al.: light absorption and scattering described by a phase function; a new approximation formula is given that is faster to compute and approximates closely the theoretical functions; use of the scattering function for simulating volume density objects (haze, fog, clouds) in a two-pass Monte Carlo ray tracing. (this paper received the best paper award).

Daldegan, et al.: physical simulation of hair on basis of a dynamic model of hair and a collision detection technique comparable to the work of Anjyo et al. Visualization with ray tracing in combination with shadow buffers.

Shirman, et al.: techniques to be used in combination with dynamic (viewpoint dependent) tessellation of curved surfaces: front- or backfacing test, light influence test, and existence of silhouette test. No direct link with ray tracing or global illumination.

Drettakis, et al.: characteristics of illumination functions (maxima, minima, critical points, etc) and application to sampling and interpolation.

Maurel, et al.: exploiting coherence in ray tracing animation sequences by keeping track of temporal invariances.

Nishita, et al.: form factor calculation for curved surface patches by surface subdivision and element projection; shadow computation using Bezier clipping.

Bao, et al.: form factor calculation for curved surface patches by approximating the surfaces with triangular elements.

Sbert: Global form factor calculation by sending a large number of rays in random directions through a scene. Discussion of estimators and error metrics.

Michelin, et al.: new form-factor calculation based on reducing the double integral to a simpler line integral; parallel implementations of the method.

Cohen, et al.: a (top-down/recursive) ray traversal method for a pyramid data structure using the midpoint line generation algorithm. Application to terrain data visualization.

***RAT*, Another Ray Tracer**, by Tom Wilson (`twilson@sunny5.dab.ge.com`)

My goal [in making my ray tracer] is very different from the others.

RAT - a ray tracing package to benchmark acceleration schemes. Although it is less primitive than its existence as my Ph.D. research work, it is still primitive compared to existing packages. It only does very basic effects (no fuzzy shadows, textures, etc.). It does contain several accelerators though: uniform subdivision, (a bad) HTE, BSP tree and octree with index grids, and extent tracing (my research). The idea of the package is to be able to plug in accelerators very easily. Someone was actually able to do this. So the accelerator, recursive bintree traversal, is also available. Bounding volumes are also selectable for further testing.

Features:

IBM PC version?	*
Amiga version?	*
Mac version?	*+

*=source code can be compiled for it. Images aren't displayed though (need to write something or convert the image)
+=I have a Mac and given time, there will be a Mac version

Sphere/Cylinder/Cone	Y
Paraboloid	Y (I've never tested it, because it's not in SPD)
Efficiency scheme?	US, HTE, BSP, Octree, extent tracing, rec. bintree

Advanced local shading - no, but it does the polygon patch shading, SPD teapot

User support	e-mail
Other S/W support	some

As you can see, it doesn't measure up to the others. But it's a research platform, not a pretty-picture generator. The accelerators should be easy to plug into existing tracers if they are modularly written. I'm currently working on an extension to my own extent tracing accelerator and struggling with the conversion of Arvo & Kirk's 5D ray classification method. I also will try to see if Ohta's tracer can be converted (if I can find it on the net again).

New object types are easily added to the code. I'd like to add new bounding volumes, but that is way down the list of things to do. Also, NFF is the input format. A very simple output format is used and a couple of conversion routines are provided (to .ppm and .ras).

I am very elated that Wim de Leeuw was able to add recursive bintree traversal to the package. He said it was pretty easy. I'm looking for other researchers to either take a stab at it or give me the code so I can try. It is far harder for me to take someone else's code that is possibly not well-structured and convert it to my package than it is for him to convert it to my documented package. It still takes a little time, but instructions are given (they are improving too), and the simplest of all examples, brute force tracing, is available as a guide. Basically, 6 routines are necessary: (1) scan the command line parameters for the technique and init. a few variables, (2) select the BVs to be used for objects (which usually just uses the defaults, but the choice is there), (3) build the technique's data structure (something the person already has done), (4) trace one ray through the structure (also already done), (5) trace one shadow ray through the structure (often this is the same as #4), (6) print diagnostics for the technique. All other code is provided (shading, intersection, etc.).

[*Contact Tom Wilson if you are interested in this research tool*]

Fast Raytracing via *SEADS*, by John Chapman (`chapman@cs.sfu.ca`) and
Wilfrid Lefer (`lefer@lifl.lifl.fr`)

Greg Stull (`gregs@irvine.com`) writes:

>IEEE Computer Graphics magazine contained an article back in the late 1980's [*actually 1986,*
>v. 6, n. 4 -EAH] which presented an approach to raytracing using a Spatially Enumerated

>Auxilliary Data Structure (*SEADS*). Other than requiring a large amount of memory, this seems
>to be a very promising approach since the rendering time of an image is "essentially" unrelated
>to the scene complexity.
>
>I am curious if anyone out there has any experience with this rendering method.

John Chapman replies:

That was Fujimoto's technique and yes it takes a lot of memory. An obvious extension is to use hierarchies of such uniform spatial subdivision structures so that 'empty' areas are not filled with huge numbers of tiny little cubes - Wyvill did some work on this with one of his students a few years ago but I can't remember exactly where it was published at the moment - try taking a look at Graphics Interface proceedings from around 90 or 91. Hope
this helps.

[*The paper he means is in GI '89, by David Jevans and Brian Wyvill, "Adaptive Voxel Subdivision for Ray Tracing" - it's a paper that should get more notice, as the technique seems quite reasonable (nesting grids, essentially). -EAH*]

Wilfrid Lefer replies:

I wrote a ray-tracing using the *SEADS* structure two years ago. Implementing this acceleration technique didn't raise any problem. You can find the C code of my software on the following ftp site:

`ftp.lifl.fr` (dir. `/pub/users/graphix/lefer/SEADS`)

Parallel Ray Tracing Schemes, by Rainer Menes
(`menes@statistik.tu-muenchen.de`)

[*This is a introduction to some of the basic problems of parallelizing a ray tracer - EAH*]

Dennis Gorrie writes:
>I'm trying to come up for a project that shows off distributed processing, and my first interest
>was ray-tracing. I don't have much background in writing rendering code, so I am not sure how
>it would be done in a distributed environment:
>
>1) render a picture 1 scanline at a time, where each machine in the pool does a scanline.
>
>2) break the picture up into n portions for n machines, and have each one work on their own
>portion.
>
>Well, I'm really not sure what the best way to do this is. For #1, this would ensure each
>machine does equal amounts of work, however, perhaps this method would be very good to use
>with an incremental integer algorithm for rendering.

Hmmm, this is not a good method to do it. You also will have to deal with the problem that the workload is not equally distributed over the scanlines. An idea which is not bad is to interleave the scanlines on each processor like this (1 Processor gets scanline 0, n, 2*n, ... the next processor get 2 get 1, n + 1, 2*n + 1, ... and so on). This gives you a static way to distribute the work over several processors. This method has two problems.

1.) It will only work for a small number of workers. Otherwise the scanlines on the processor are so far from each other that they are not related and don't have the same amount of work, and a 640x480 picture can only be split into 480 working packages.

2.) Antialiasing is hard to use. Normally antialising uses the information of the lines calculated before on a single processor machine. On a network of processors you don't have that information (or you will need a lot of communication). To do antialiasing for the worst case you have to calculate the scanline before you and the scanline which follows you on each processor. This will raise your work amount by 3 times!

>Method #2 might get around this problem, and reduce network traffic; however, the problem
>arises that some areas of the picture are more complex than others, so while some machines are
>already done their trace and sitting idle, other machines are still grinding away.

Not bad if you use square regions like 8x8 or 16x16 pixels with dynamic workloading. At startup every processor requests one block of pixels to work on. When he is done he will send his result back to the loadbalancing process and get a new pixel block package to work on. This gives you a good loadbalancing of the net and only introduces a small amount of overhead when using antialising. For antialiasing of 16x16 blocks you have an overhead of about only 1/3 and not 3 times. If you make the areas bigger this will make your overhead smaller but reduce the amount of packages. To get a good distribution of work you will need 20-30 packages per processor.

>It would be nice if I had a basic rendering function that I could pass a list of objects and a
>scanline #, and have it generate the scanline. Then I could concentrate on the task of splitting
>up the job.
>
>Perhaps another aspect is a shared data file which all the tracers access, maybe this would speed
>up the algorithm. I don't know.

All this is based on my implementation of POVRay 1.0 (soon POVRay2.0) and Rayshade 4.06 onto a transputer workstation with up to 129 workers. The architecture is able to work with up to 526 workers.

Notes on Parallel Ray Tracing, by Hsiu Lin (`hlin@wsuaix.csc.wsu.edu`) and
Sam Uselton (`uselton@nas.nasa.gov`)

[*From the massively parallel rendering mailing list. There were many comments about these questions, I chose Sam's as a good, broad response, and also show Lin's responses to that. Subscribe to the mailing list by writing* `mp-render-request@icase.edu`. *-EAH*]

Hsiu Lin writes:
>>1. Are there still many groups or people interested or doing research on parallel raytracing?

Sam Uselton responds:
>Ray tracing is "embarrassingly parallel," in fact, I believe the first time I read the term (in an
>article by Gordon Bell) ray tracing was used as the example to explain the concept.
>"Embarrassingly parallel" applications are excluded from the Gordon Bell Award.

Lin replies:

Yes, it is a embarrassingly parallel problem. But, since pixel computations might vary significantly in all screen pixels, it will result in poor results without careful scheduling. In parallel computing literature, you also can find lots of paper talking about how to schedule this kind embarrassed paralleling problem (even though their problem is not ray tracing). Check IEEE Transactions on Computer, Dec. 1987 "guided self-scheduling" or "a practical and robust method for scheduling parallel loops" in Comm. of ACM,
Aug. 1992.

>Basically, if each pixel (or set of pixels) is computed by a single process (that is, no single pixel
>receives contributions computed by several processes) then there are no synchronization
>problems and no data dependency problems.

Yes, no data dependency. But, as for synchronization, it depends on the algorithm implemented. For a simple example, we can use master/slave model to parallelize ray tracing. Each slave requests a job from the master if it needs one. If the system is small, it works fine. But, if we talking about 200 or even 1024 slave nodes, the story is different. The large synchronization overhead occurring in the master node will degrade the performance.

>Therefore, simply porting a ray tracer to a parallel machine is not very interesting as a
>research project. Research must involve some unobvious cleverness to exploit the parallel
>architecture.

I can not completely agree with this point. How you can achieve a scalable and load balanced parallel raytracer (if we talking about big problem)? For example, millions of polygons are raytraced on a parallel machine.

>>2. Is parallel raytracing too simple or trivial?

>See above.

>>3. Is parallel raytracing research dead?

>Not entirely, but it has been worked enough that good ideas are becoming harder to find.

>>4. Many previous work used SPD to test parallel raytracing. Is it fair? Although an SPD
>>database (such as balls, tree, mountain etc.) can be complicated enough, most objects appear
>>uniformly in the scenes. I doubt if it will make the tested scene itself easy to be load balanced.
>>And thus it is very easy to use many different schemes to get very good speedup. Why don't
>>we view these scenes in different ways? It might result in more difficult scene to be load
>>balanced.

>SPD is a start, but a wider variety of scenes would seem to me to provide more info. But
>designing good test scenes requires much time and effort for little reward.

I don't mean SPD is not good. Actually, I enjoy using it very much. I just mean we need to adjust the viewpoint to keep the objects appear in different locations of the screen (i.e. more different pixel computation distributions on the screen) to test the effectiveness of parallel ray tracing. Otherwise, it is very easy, just like you think, to parallelize raytracing a self-balanced scene using different schemes.

[*This parallels some of my comments to Erik Jansen. -EAH*]

>>5. People said now most interesting problem in parallel raytracing is how to partition

>>database and distribute database.(if database is too large to duplicate at each node.) Since we
>>raytrace huge database, it should need huge cpu time to finish. When we compare this cpu
>>time with remote data access time(when database is distributed), maybe the data access time is
>>relative small. I doubt if this issue will become the same as we only consider all nodes has all
>>duplicated database.

>If the database is replicated, or if you are using a shared memorymachine, then you should get
>linear speed up as you add processors. If you aren't, you should instrument your system and
>find the bottleneck(s). Look at the papers in the Parallel Rendering Symposium and you will
>see that distributing too small a chunk of work is inefficient, but that for reasonably sized tasks,
>communication time can easily dominate compute time, when the data must be distributed.
>Distribution among workstations on an ethernet is fine to reduce runs that require days to runs
>that require hours, but you can never get interactivity that way.

>>6. Besides loadbalancing and data partition, are there any other important issues? I have one.
>>How to predict your good speedup will drop as the number of processors increase.

>This issue is very architecture and implementation dependent. If you are doing a good job of
>data partitioning and load balancing, that WILL mostly minimize the loss of efficiency.

Yes, I believe you are right. But, to design a good parallel raytracer, we can not ignore the architecture issue (particularly when we are talking parallel computing). So, how can you easily do a good job on both above issues?

Well! How about parallel volume rendering using ray casting? In my opinion, volume rendering uses more regular data and regular computation than ray tracing. Do you agree? If so, do you also think parallel volume rendering is much easier than parallel raytracing?

To Lin's comments, Uselton responds:

I didn't mean to indicate that I thought that all issues around parallelizing ray tracing were solved. What I want to emphasize is that, if I'm on a conference program committee, and I see a paper whose total content is essentially:

> "I parallelised my ray tracer and got near linear speed up on machine x," and maybe some words about whether it is SIMD or message passing, tightly coupled or network connected machines, etc.

I'm not very interested in the paper.

If it includes something about how load balancing and/or distribution and access of very large data bases then it has a better chance. But what I'm really interested in at this point is work that performs some detailed measurements and varies some parameters to get data on how performance varies in response to the parameters. And it would be even better if there is some cogent analysis as to why this connection should be true.

As I argued in a paper given before a small conference about 8 or 10 years ago, the question is NOT "can ray tracing be parallelized?" but rather "which of the myriad possible ways (that will ALL work to some extent) is the best, and under what particular circumstances is this true?"

I think much of the ray casting volume rendering work is directly applicable to ray tracing as well. I think more research institutions are pursuing volume rendering because its applications make it more fundable. Let me point again (but more specifically) to David Ellsworth's paper in the Parallel Rendering Symposium. It is not about ray tracing, but it is exactly the kind of analysis that could be done for ray tracing (and many other parallel applications).

I hope this clarifies my position (and restores my credentials as a ray tracing enthusiast - parallel and otherwise).

Parallel Texturing, by Jon Leech (`leech@cs.unc.edu`) and Brian Corrie (`Brian.Corrie@cs.anu.edu.au`)

Jon Leech writes:

> We're implementing procedural shading models (ala RenderMan) to run on the SIMD
>rasterizer/shader arrays in Pixel-Flow, the next generation of graphics system being built here.
>The SIMD processors are coupled to a heavily interleaved memory for fast image texturing
>with trilinear interpolation.
>
> I'd say one of the main problems is how to make efficient use of the SIMD processors by
>scheduling identical operations in different shaders (being applied at disjoint pixels) to execute
>at the same time. This is important when using lots of shaders in the same image.

We have done the same thing for our parallel volumetric/geometric renderer for the Fujitsu AP1000. That is, we have implemented an extended RenderMan-like shading language for procedural texturing. The extension encompasses procedural shading of a volume data set. To do this, we have created a new class of shader (called a _data shader_) that determines the shading and attenuation applied at a sample point within the volume. The shader is invoked at each sample point.

Our implementation of the shading language gives us procedural shading of surface primitives as well. With respect to the parallel issues of our technique, I can safely say that we have addressed approximately 0% of them 8-). The data shaders were implemented as research into volume rendering techniques, not parallel rendering. It just so happens that we have implemented the technique on our AP1000 as well. Whether we go further with this research with respect to parallelism is unclear at this time.

If you are interested in more information on this work, see our paper entitled ''Data Shaders'' in the Proceedings of Visualization '93, San Jose, California, October 1993.

You can also check out our tech report ''Data Shader Language and Interface Specification'', Technical Report TR-CS-93-02, Department of Computer Science, Australian National

University. It is available for anonymous ftp from `dcssoft.anu.edu.au` in the directory `pub/techreports/tr-cs-93-02/report.ps.Z`

And if you are really keen, or just curious about the ANU ParVis Visualization Project, you can access an ftp based www server (with NCSA mosaic for example) on the CAP research project, including the ParVis Visualization Project, at the URL file:`//dcssoft.anu.edu.au/pub/www/dcs/cap/cap_research.html`. Info on the ParVis project can be found by following the ‘‘ParVis Visualization Project’’ link at the bottom of this document.

Mapping Texture on a Sphere, by Ron Capelli (`capelli@vnet.IBM.COM`)

D.J.James writes:
>I haven't had much success trying to wrap a texture around the sphere. My first approach using
>atan2() produces huge distortions at multiples of 90 degrees longitude. I tried using spherical
>map projection equations using asin() and acos() without any luck. The best shot I had using an
>equation from a gl reference book produced a beautiful image on the front of the sphere but the
>rear kind of swirled into a black hole.
>
>Can someone give me a clue as to how to wrap an image around a sphere?

"You can't comb the hair on a billiard ball."

I may be mis-remembering the above quote slightly, and can't find the original source, but it refers to the problem of oriented mappings on a sphere.

Anyway, there was a paper presented at SIGGRAPH '93 that addresses the problem of minimizing texture mapping distortion:

Maillot, Yahia, and Verroust, "Interactive Texture Mapping", ACM/SIGGRAPH

Computer Graphics Annual Conference Series, Proceedings of SIGGRAPH 93 (Anaheim, CA, Aug.1-6, 1993), pp.27-34.

The problem was treated as a global energy minimization problem, where energy was related to elastic deformation of the texture map.

Other very good papers addressing the problem in different ways are:

Bennis, Vezien, and Iglesias, "Piecewise Surface Flattening for Non-Distorted Texture Mapping", ACM/SIGGRAPH Computer Graphics, Vol.25, No.4, July 1991, Proceedings of SIGGRAPH 91 (Las Vegas, NV, July 28 - Aug. 2, 1991), pp.237-246.

Bier and Sloan, "Two-part texture mapping", IEEE CG&A, Vol. 6, Sept. 1986, pp.40-53.

Computational Geometry in C, by Joseph O'Rourke
(`orourke@sophia.smith.edu`)

[*This looks like an interesting book; I've ordered a copy and hope to review it in the future. -EAH*]

author =	"Joseph O'Rourke"
title =	"Computational Geometry in {C}"
publisher =	"Cambridge University Press"
year =	1993
note =	"In stock on 28 January 1994. ISBN 0-521-44592-2/Pb \$24.95, ISBN 0-521-44034-3/Hc \$49.95. Cambridge University Press, 40 West 20th Street,

New York, NY 10011-4211.
1-800-872-7423."
note = "346+xi pages, 228 exercises, 200 figures, 219 references"
comments = "Chapter titles:

1. Polygon triangulation
2. Polygon partitioning
3. Convex hulls in two dimensions
4. Convex hulls in three dimensions
5. Voronoi diagrams
6. Arrangements
7. Search and intersection
8. Motion planning
9. Additional topics"

annote = "Textbook"

Programming for Graphics Files in C and C++, by John R Levine (`johnl@chico.iecc.com`)

John Levine, **Programming for Graphics Files in C and C++**, John Wiley, ISBN 0-471-59856-9 (with disk), ISBN 0-471-59854-2 (no disk). $29.95 without disk, $49.95 with disk.

The book discusses programs that read and write graphics files in a variety of formats. The book is about 60% C and C++ code, much based on the well-known PBMPLUS library, the Leffler TIFF library, and the Independent JPEG group's JPEG software. (The disk has the full source of all three, the book is just the most interesting parts, since otherwise it would have been about 2000 pages long.) There is also new code for C++ image memory management in limited memory environments like DOS, and some new file management code to read formats not handled by PBM such as HP-GL files and Windows BMP. As far as I am concerned, all of the code is freely redistributable.

After discussing general principles of graphics file wrangling, the book exhibits and explains the code to read and write 13 formats.

It's available at bookstores, or direct from Wiley at 1-800-879-4539.

Point in Polygon, the Quick Answer, by Wm. Randolph Franklin and Eric Haines

Wm. Randolph Franklin (`wrf@ecse.rpi.edu`) posted a nine line algorithm for the classic question "How do I detect if a point is inside a polygon?" I made a few minor speed hacks, got rid of a little code, and made it shoot a ray along X+ (which I'm used to), but it's mostly his. Since we've beaten this one to death in these pages, I thought I'd give his nice short (though a tad dense it's mostly just one big if statement) answer (8 lines). Here 'tis:

```
int pnpoly(int npol, float *xp, float *yp, float x, float y)
{   int      i, j, c = 0;
    for (i = 0, j = npol-1; i < npol; j = i++) {
        if ((((yp[i]<=y) && (y<yp[j])) || ((yp[j]<=y) && (y<yp[i])))
            &&
        (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))
        c = !c;      }
    return c;  }
```

Running it in my pt-in-poly testbed I found this routine to be about 20% slower than my optimized crossings test (in the upcoming Graphics Gems IV).

Elib and NetNews Information, by Eric Haines

There is an interesting pair of resources I just found out about from "Wired" magazine. By writing <elib@DB.Stanford.EDU> and <netnews@db.stanford.edu> and sending the message "help" (no subject) to each, you can find out about them, too. Elib lets you search a computer science reference database at Stanford for keywords. It's not very extensive (dating back to 1991 at best), but the response is immediate and the price is right (free). For example, it turned up this, which I hadn't heard of:

Author :	Garland, M. J.
Title :	Portable parallel ray tracing algorithms.
Institute :	Carnegie-Mellon University. Department of Computer Science.
Report :	CMU-CS-93-176.
Date :	1993.
Keyword :	Realistic rendering.
Keyword :	Ray tracing.
Keyword :	Parallel ray tracing.

NetNews is like Elib, but the server searches recent News on the net using keywords. You can also set searches on keywords so that it will daily (or at whatever frequency you want) send you the first part of any messages that match. I do recommend that you set the threshold to 60, as the default is 50 (not 70, as it says in the documents). At 50 and the keyword "ray tracing", I was getting matches on rebuilding engines, how much LSD Jerry Garcia uses, abortion discussions and all kinds of other random stuff - amusing for a few days, but it wears thin. Anyway, as an example, here's one that NetNews found today that I would have never run across otherwise:

>From: pjg@parint.esl.com (Paul Gyugyi)
Newsgroups: rec.toys.lego

dpenney@morgan.ucs.mun.ca (Dave Penney) writes:

> 2a) If I was to design my own Lego set from Lego standard pieces, where would I be able to post this for others? (FTP, Pictures, or would LEGO be willing to take it?)

2b) How (or which would be the easiest way) to transcribe this down onto paper?

There are a variety of CAD tools for lego on the earthsea archive, but nothing that doesn't require some work. If you can sketch down your instructions and fax them to me, I'll pass them along to other CAD developers to use as a test bed for adding new features to the programs. In earthsea:/pub/lego/CAD, you will find some Postscript tools, some ray tracing tools, and some parts libraries for drafting programs. But I've found isometric graph paper (with 30 degree lines) is easiest to use, actually. We've been working on a standard input file format, something that you can type in ASCII and feed into other programs.

Stefan and I are working on a front end to "Rayshade", a 3D raytracing program. If you are interested in describing your models in our format, or if you are willing to sit down with a ruler and add descriptions of some lego pieces to our parts library, it would be appreciated.

CFP: 5th Eurographics Workshop on Rendering, by Georgios Sakas

Darmstadt, Germany 13-15 June 1994

Aims and Scope: Following four successful workshops (Rennes-1990, Barcelona-1991, Bristol-1992, Paris-1993) we announce the fifth workshop on rendering techniques. In the recent years the workshop has been well established as a major international forum in exchanging experience and knowledge between people from universities, research and industry interested in the different aspects of rendering techniques.

Main topics include (but are not restricted to):

- Radiosity
- Ray tracing
- Illumination models

- Colour, texture
- Sampling, filtering, anti-aliasing
- Parallel solutions for rendering

Two special themes of this workshop are:
- Illumination & rendering of participating media (volume objects, clouds, ...)
- Rendering of architectural & CAD models (illumination simulation, real-time rendering, walkthroughs, handling of large datasets, ...)

We encourage also papers describing on-going research and providing new techniques, perspectives and applications in the field. Discussions and evaluation of current techniques and future trends greatly contribute to the attractivity of the workshop. Thus, the presentation of the papers to the plenum will be followed by a discussion introduced by an expert of the field. Internationally renowned speakers from research and industry will also present invited papers.

Schedule: 5 April	1994	Submission deadline
5 May	1994	Notification of acceptance for presentation
30 May	1994	Full-paper deadline for the workshop proceedings
13-15 June	1994	Workshop

For [the full posting,] detailed information, rendering databases, and authors toolkit including latex styling formats contact `haas@igd.fhg.de`, `gsakas@igd.fhg.de` or `shirley@cs.indiana.edu`

CFP: 5th EG workshop on Animation & Simulation, by Gerard Hegron

(`hegron@irisa.fr`)

Oslo, Norway 17-18 September 1994

AIMS AND SCOPE:

The Eurographics workshop on Animation and Simulation has become an international forum of high quality for exchanging experience and knowledge between people representing the animation and simulation communities on the general themes of modeling, animation, motion control, simulation and visualization of dynamic scenes.

SCHEDULE:

May 1 : deadline for extended abstract

June 1 : notification of acceptance for the workshop

July 15 : full papers

September 17-18 : workshop

Write Gerard or Olov Fahlander (`olov-f@isy.liu.se`) for the full posting.

Morphology Digest, by Henk Heijmans (`henkh@cwi.nl`)

Some time ago we announced the "Morphology Digest", an electronic newsletter for workers in the field of mathematical morphology, stochastic geometry, random set theory, image algebra, etc. It provides fast and up-to-date information on:

1. Conferences, workshops, courses
2. Books, articles, reports, PhD theses
3. Algorithms, software, hardware
4. Available research positions
5. Bibliographical data

relevant to anyone interested in mathematical morphology.

The digest appears every six weeks and has approximately 500 subscribers.

Subscribe: email to morpho@cwi.nl with "subscribe" as subject and empty message body.

Unsubscribe: email to morpho@cwi.nl with "unsubscribe" followed by email address as subject and empty message body.

Submissions: email to morpho@cwi.nl with "submit" as subject.

Archive site: anonymous ftp to ftp.cwi.nl [192.16.184.180]; directory /pub/morphology/digest

[*This digest is not about morphing, it is much more oriented to image processing, object recognition, texture classification, etc. It's a pretty serious (and useful) journal, if this area interests you. -EAH*]

Position of the Sun, by Joe Cychosz (3ksnn64@ecn.purdue.edu)

I didn't try to analyze [James Ashton's] code [in RTNv6n3 for sun position]. But, I assume it's a quick hack. It doesn't consider latitude, for one thing. I didn't check his time equations, but there is nothing to correct for timezone. Maybe he's working with GMT all the time and what is this floating point month stuff? The lengths of the months varies from month to month.

The following code computes the direction to the sun, give location on the earth (lon, lat) and the time. A correction angle (gamma) can be specified so the sweep of the sun can be aligned with the world coordinate system. Thus the structure can be nicely modeled along the coordinate system. Some years ago I did a short film of an energy efficient house build here at Purdue. One of the key pieces of technology was the design of the overhangs over the windows which were designed to keep the sun out in the summer and let it in in the winter. The film demonstrated the effects of the overhangs on the penetration of sun into the house at various times of the year.

Hopefully the code has sufficient documentation to enable one to use. This code has only been tested in the northern hemisphere. A quick note about the coordinate system: y is up, x is right, z is out.

Joe Cychosz

```
#include     <math.h>
#include     "GraphicsGems.h"

#define pi   3.14159265358979323846264338
#define      rad2deg   (180./pi)
#define      deg2rad   (pi/180.)

/* ----      trigd.h - Degree based trig functions. ------------      */

#define      cosd(t)            (cos ( 0.01745329251994 * (t) ))
#define      sind(t)            (sin ( 0.01745329251994 * (t) ))
#define      tand(t)            (tan ( 0.01745329251994 * (t) ))

/* ----      sunpos - Compute the position of the sun. ---------      */
/*                                                                   */
/*                                                                   */
/* Description:                                                      */
/*  Sunpos computes the position of the sun given the location       */
/*  (longitude and latitude) on the earth, the day of the year,      */
/*  and the time.  The origin (i.e. 0,0,0 in world space) is         */
/*  the specified location on the earth.  With an orientation        */
/*  angle of 0, east = -z, west = +z, south = +x, north = -x.        */
/*                                                                   */
/* On entry:                                               */
/*     lon, lat= The longitude and latitude in degrees of the        */
/*          location of the origin on the earth.           */
```

```
/*          (i.e., West Lafayette, Indiana = 89-40)                   */
/*     lontz   = The longitude for the time zone in degrees.          */
/*          Each time zone is 15 degrees, advancing to the  */
/*          west.  For daylight savings, the longitude is   */
/*          advanced 1 timezone to the east.                          */
/*          (i.e., 75 for EST, 90 for CST, 75 for CDT)                */
/*     time    = The time of the day in hours.                        */
/*          (i.e., 13.5 = 1:30 pm)                                    */
/*     day     = The day of the year.                                 */
/*          (i.e., 172 for June 21st)                                 */
/*     gamma   = Orientation angle defining south, rotating the      */
/*          sweep of the sun.                                         */
/*          (i.e, 14 = 14 degrees east of south)            */
/*                                                                    */
/* On return:                                                         */
/*     pos     = Direction cosines of a vector pointing toward       */
/*          the sun from the origin.                                  */
/*                                                                    */
/* Returns:  True if the sun is visible.                    */
/*                                                                    */
/* Author:     Joe Cychosz                                            */
/*      Purdue University CADLAB                                      */
/*      Potter Engineering Center                                     */
/*      W. Lafayette, IN  47907                             */
/*                                                                    */
/* ------------------------------------------------------------      */


boolean     sunpos            ( lon, lat, lontz, time, day, gamma, pos)

   double            lon, lat; /* Longitude, latitude       */
   double            lontz;             /* Longitude for timezone    */
   double            time;              /* Time of day               */
```

```
   int                  day;                /* Day of the year */
   double               gamma;              /* Orientation angle        */
   Vector3              *pos;               /* Position vector of sun   */

{           /* Angles are in degrees, times are in hours.   */
   double               del;                /* Angle of declination     */
   double               b;                  /* Time angle               */
   double               e;                  /* Time                     */
   double               w;                  /* Hour angle               */
   double               we, gs;
   double               phi;
   double               ws;                 /* Sunrise, sunset angle    */
   double               sunrise;            /* Sunrise time             */
   double               sunset;             /* Sunset time              */

/* Compute solar declination (23.45 deg on June 21)                     */
   del = 23.45 * sind (360.0 * (284+day) / 365);

/* Compute the hour angle in degrees.                                   */
/*                                                                      */
/* w = 0 at solar noon, => 180 at midnight, + for am, - for pm.         */
/* 1 hour = 15 degrees, w = 22.5 at 10:30 am (solar)                    */
   b   = 360.0 * (day-81) / 364;
   e   = (9.87*sind(2.0*b) - 7.53*cosd(b) - 1.5*sind(b)) / 60.;
   w   = 180.0 - lontz + lon - 15.0*(time+e);

   phi = acos (sind(lat)*sind(del) + cosd(lat)*cosd(del)*cosd(w));
   we  = acos (tand(del) / tand(lat)) * rad2deg;
   gs  = asin (cosd(del) * sind(w) / sin(phi));
   if  ( w >  we ) gs =  pi - gs;
   if  ( w < -we ) gs = -pi - gs;
```

```
/* Compute the position of the sun.                                    */
/*                                                                     */
/* For gamma = 0, the sun will rise at -z and set at +z.               */
   pos->x =   cos (gs + gamma*deg2rad);
   pos->y =   cos (phi);
   pos->z = - sin (gs - gamma*deg2rad);

/* Compute sunrise, sunset angle.                                      */
   ws = rad2deg * acos (-tand(lat) * tand(del));

/* Compute sunrise time and sunset time.                               */
   sunrise = (180.0 - ws - lontz + lon) / 15.0 - e;
   sunset  = (180.0 + ws - lontz + lon) / 15.0 - e;

   printf (" sunpos: sun position: %f %f %f\n", pos->x, pos->y, pos-
   >z);
   printf (" sunpos: sunrise time: %f\n", sunrise);
   printf (" sunpos: sunset  time: %f\n", sunset);

   return (time >= sunrise && time <= sunset);
}
```