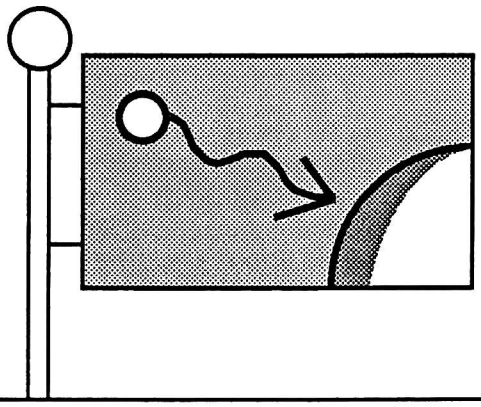
The Ray Tracing News



Volume 2, Number 1 "Light Makes Right" February 1988

Feature Articles

1 Problems with the Octree Subdivision Method for Ray Tracing ____ *Eric Haines*

3 Editorial: Optical Junk and Rendering Times ____ *Andrew Glassner*

4 Notes Regarding Ray Tracing withOctrees ____ *Olin Lathrop*

4 Minimal Ray Tracing ____ *Paul Heckbert*

6 Subspaces and Simulated Annealing ____ *Jim Arvo*

8 Top 10 Hit Parade of Computer Graphics Books ____ *Eric Haines*

9 A Rendering Trick & Puzzle ____ *Eric Haines*

10 Efficiency Tricks ____ *Eric Haines*

11 Efficient Ray Tracing in Subdivided Space ____ *Andrew Glassner*

Andrew Glassner
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175 USA
(919) 962 - 1803 / glassner@cs.unc.edu

Problems with the Octree Subdivision Method for Ray Tracing

by Eric Haines

I think that octree efficiency techniques can be terrible for generalized ray tracing. The technique is useful for a large percentage of cases, but there exist pathological environments which can cause serious problems. The root of these problems is that there is no guarantee that a leaf node in an octree will have a reasonably low number of primitives to test.

There are two limits imposed by the programmer on how an octree is formed for a given environment. One limit is the number of primitives in a octree leaf node (a leaf node is an octree voxel which contains a list of primitives which are to be tested for ray intersection. Parent nodes are voxels that are subdivided into subvoxels). The other limit is the number of levels to which the octree subdivision can be taken. A simple method of generating the octree structure is then to check how many objects are in a leaf node. If this exceeds the limit, the octree node is subdivided, new lists of primitives for the eight subvoxels are created, and each subvoxel is tested in the same manner, recursively. When a leaf node is at the maximum subdivision level it is not subdivided, no matter how many objects are contained within.

Problems with the octree begin to arise when

(continued on Page 2)

Problems with Octrees
(continued from Page 1)

primitives do not fill space with the same density. For example, imagine you have a football stadium made of, say, 5000 primitives. Sitting on a goal line is a shiny polygonalized teapot of 5000 quadrilaterals (note that the teapot is teapot sized compared to the stadium). You fill the screen with the teapot for a ray trace, hoping to get some nice reflections of the stadium on its surface.

If you use an octree for this scene, you'll run into an interesting problem. The teapot is, say, a foot long. The stadium is 200 yards long. So, the teapot is going to be only 1/600th the size of the stadium. Each octree subdivision creates 8 subvoxels which are each half the length of the parent voxel. You could well subdivide down to 9 levels (with that 9th level voxel having a length of 1/512th of the stadium length: about 14 inches) of octrees and have the whole teapot inside one octree voxel, still undivided. At best you will have subdivided the teapot only once. If you stopped at this 9th level of subdivision, your ray trace would take forever. Why? Because whenever a ray would enter the octree voxel containing the teapot (which most of the rays from your eye would do, along with the reflection and shadow rays), the voxel would contain a list of the 5000 teapot polygons (plus the stadium's playing field polygon). Each of these polygons would have to be tested against the ray, since there is no additional efficiency structure to help you out. In this case the octree has been a total failure.

Now, you may be in a position where you know that your environments will be well behaved: you're ray tracing some specific object and the surrounding environment is limited in size. However, this answer certainly does not help the designer who is attempting to create a system which can respond to any user's modeling requests. Further subdivision beyond level nine down to (say) level eighteen may solve the problem in this case. But I can always come up with a worse pathological case. A realistic example is an animation of a satellite orbiting the earth: the sphere which represents the earth would create a huge octree node, and the satellite would easily fall within one octree cubie (I compute that you'd have to go down about 23 levels for a five foot long satellite to begin subdivision). Or a user simply wants to have a realistic sun, and places a spherical light source 93 million miles away from the scene being rendered. Ridiculous? Well, many times I find that I will place positional light sources quite some distance away from a scene, since I don't really care how far away the light is, but am interested in only the direction the light is coming from. If a primitive is associated with that light source, the octree suddenly gets huge. My final example is of a teapot in a room. Here we find that the first four levels of subdivision are used just to get us to the level of the teapot. These extra subdivisions cost us extra time and memory for the octree, yet provide very little benefit. Many empty octree voxels may have to be traversed when there are large amounts of space between objects.

Solutions? Mine is simply to avoid the octree altogether and use Goldsmith's automatic bounding volume generation algorithm (IEEE CG&A, May 1987). However, I hate to give up all that power of the octree (i.e. if any primitive in an octree voxel is intersected, then ray trace testing is done, versus having to test the whole environment's hierarchy against the ray). So, my question: has anyone found a good way around this problem? One method might be to do octree subdivision down to a certain level, then consider all leaf voxels that have more than the specified number of primitives in their lists as "problem voxels". For this list of primitives we perform Goldsmith's algorithm to get a nice bounding volume hierarchy. The idea is to use the octree for the total environment so that the quick cutoff feature of the octree can be used. Using bounding volume hierarchy locally gets rid of the pathological cases for the octree.

"Problem voxels" could probably best be identified by checking the longest list size among the subvoxels with the parent voxel's original list size. If the subvoxel list length is not noticeably smaller (say at most 60% of the parent list length), then the octree subdivision is deemed unnecessary and is not done. This test would catch the case where a subdivision operation does not gain very much for overall testing.

The above procedures would help cure the pathological cases discussed, though at a loss of elegance. Has anyone else considered this problem and found some alternate solution? ●

Editorial: Optical Junk and Rendering Times

by Andrew Glassner

As I walked past a stationary store last December, I noticed the Christmas tree ornaments. Amidst the blinking lights and spinning wooden Santas and trains were reflective ball ornaments – the shiny spheres so well known to every ray tracer!

I wonder how many of us have actually gone out and bought Christmas tree balls to stare at and study. I remember when I was first learning about colored lights on colored surfaces. I found a post-Christmas sale of six shiny spheres of different colors, and placed them on my desk. I learned a lot about light, color, and reflection by observing those shiny globes.

Reflections are neat, and they certainly make reflective surfaces look reflective. But just because we can do reflections doesn't mean we have to. In his book *The Visual Display of Quantitative Information* (Graphics Press), Edward Tufte refers to needless ornament in a chart as "chart junk". This sort of thing can arise in many ways, but I find I'm most susceptible to including chart junk when I'm working with a computer, say in a program like MacDraw. It's just so easy to pile in layer after layer of texture and ornament that it becomes almost a game, to the point where I sometimes forget that I'm making a chart to convey information, not dazzle the eye.

I think in a similar vein we're susceptible to including "optical junk" in our ray-traced images, when we throw in extraneous reflective and refractive objects just because we can. Looking around our graphics lab right now, I see almost no transparent or reflective objects, except for the faces of CRT screens, the windows in the doors, and the eyeglasses on my head. Well, the coffee mug next to me is glazed and has some very low-level reflection, but it's not very significant. Most man-made interior environments reflect diffusely, I believe. It's primarily shadows and textures that make a scene look good, not lots of bouncing light.

Of course when we do have a reflective or refractive surface then that little bit of realism might help an image tremendously, but such objects are not as prevalent as most of our recent ray-traced images would imply. I'm as guilty of optical junk as anyone, but I'm trying to learn restraint.

One of my New Year's resolutions is to not use comparitive running times to evaluate competitive ray tracing algorithms. At the Siggraph '88 ray tracer's roundtable there was a discussion of how to actually compare ray tracing algorithms, and we never really got anywhere.

I think one of the best steps in this direction is Eric Haines's package of standard databases. They give us something to grab hold of, and they're also attractive images that are a pleasure to render. But it's not the running time that should be observed when rendering these databases, it's the relevant statistics that describe the work done by the algorithm.

For example, suppose I have a brute-force ray tracer, which tests every ray against every object. The program is small and the inner loops are fast. Of course, they repeat a zillion times, but the program is short. On the other hand, consider a sophisticated ray tracing system, which supports procedural objects and textures, distributes the rays every which way, and manages the database in an object-oriented manner (which we all know can means lots of little procedure calls, for everything from dot product to intersection). For some databases, it is possible that the brute-force program will actually run faster than the sophisticated system, and the images might even be of comparable quality!

So unless we can compare two algorithms side-by-side in completely equivalent environments (by this I mean the program environment; everything the same but one or two procedures), then clock times will be meaningless at best and misleading at worst. It is my belief that such side-by-side comparisons are so hard to build that it's just not worth the effort. How do you put pure bounding volumes and pure space subdivision side-by-side?

One suggestion from the roundtable was to build bare-bones renderers that shot one eye ray per pixel, computed only gray scale, and so forth. Perhaps this could work, but the mechanics are hard. If I'm working on algorithm A, I'm likely to be much closer to the efficiency

(continued on Page 4)

Editorial

(continued from Page 4)

tradeoffs and coding strategies for A than for algorithm B. Even if I try to give B a good implementation, my code for A is likely to be better. And if I use someone else's code for B, then I think any meaningful comparisons are immediately lost.

So since we don't really know how to compare ray tracing algorithms at this point, let's not mislead ourselves by doing it anyway. Leave clock times out of it. Report the relevant statistics that describe the performance of the program in raw terms: number of ray/object intersections, number of rays, some accounting for any pre-processing work, number of ray/bound intersections for bounding volumes, number of cell traversals for space subdivision, and so on. Discuss the results and mention where the big effects are visible, and how big they seem to be, and let the reader agree or disagree that an important cost has been reduced. But don't then produce the elapsed running time for making the picture on your system as a final justification; we want to learn about algorithms, not who's the best hacker! ●

Notes Regarding Ray Tracing with Octrees

by Olin Lathrop

1) I don't use Andrew's hashing scheme to find the next voxel when tracing rays in adaptively subdivided space (IEEE CG&A, October 1984). Instead, I transform the ray so that my octree always lives in the (0,0,0) to (1,1,1) cube. To find the voxel containing any one point, I first convert the coordinates to 28 bit integers. The octree now sits in the 0 to 2^{28} cube. Picking off the most significant address bit for each coordinate yields a 3 bit number, which is used to select one of 8 voxels at the top level. Then I pick off the next address bit down to select the next level of subordinate voxel, and so on with increasingly less significant bits, until I hit a leaf node. This process is log(n) in theory, and very quick in practice. Finding a leaf voxel for a given integer coordinate consumes less than 10% of the total rendering time for all of my images so far. I store direct pointers to subordinate voxels directly in the parent voxel data block. In fact,

(continued on Page 5)

Minimal Ray Tracing

by Paul Heckbert

A complete ray tracer in the C programming language!

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*
A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s
->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l->sph)if((e=1
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*
eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*pixar!ph*/
```

Octree Notes

(continued from Page 4)

this is the **only** way I have of finding all but the top voxel.

About the 28 bit integers: At first glance, it would appear some precision is being thrown away by converting to 28 bit integers, consider that the original floating-point number occupies 32 bits. Actually, the 32 -bit floating point numbers only have 24 bits of significance anyway. 28 bits is a comfortable precision that introduces very little additional error, and still leaves some overrange, since it is stored and manipulated as a 32 bit integer.

2) Choosing subdivision criteria: First, the biggest win is to subdivide on the fly. Never subdivide anything until you find there is a demand for it. Here are my current subdivision criteria in order of precedence (I overrides II) :

I) Do not subdivide if the generation number of this node is at the subdivision generation limit. I think everyone does this.

II) Do not subdivide if the voxel is empty.

III) Subdivide if the voxel contains more than one object.

IV) Do not subdivide if less than N rays passed through this voxel, but did not hit anything. Currently, I set N to 4.

V) Subdivide if M*K < N. Where M is the number of rays that passed through this voxel that **did** hit something, and K is a parameter you choose. Currently, I set K to 2, but I suspect it should be higher. This step seeks to avoid subdividing a voxel that may be large, but has a good history of producing real intersections anyway. Keep in mind that for every ray that did hit something, there are probably light source rays that did not hit anything. (the shader avoids launching light rays if the surface is facing away from the light source). This can distort the statistics, and make a voxel appear less "tight" than it really is, hence the need for larger values of K.

VI) Subdivide.

Again, the most important point is lazy evaluation of the octree. The above rules are only applied when a ray passes through a leaf node voxel. Before any rays are cast, my octree is exactly one leaf node containing all the objects.

3) First solution to teapot in stadium: This really cries out for nested objects. Jim Arvo, Dave Kirk, and I submitted a paper last year called *The Ray Tracing Kernel* which discussed applying object oriented programming to the design of a ray tracer. Jim has an article in this issue that goes into more detail, so I will make this real quick. Basically, objects are only defined implicitly by the results of various standard operations they must be able to perform, like "intersect yourself with this ray". The caller has no information **how** this is done. An object can therefore be an "aggregate" object which really returns the result of intersecting the ray with all its subordinate objects. This allows for easily and elegantly mixing storage techniques (octrees, linear space, 5D structures, etc.) in the same scene. For more details, see Jim Arvo's article in this issue.

4) Second solution to teapot in stadium: I didn't understand why an octree wouldn't work well here anyway. Suppose the teapot is completely enclosed in a level 8 voxel. That would only "waste" 8x8=64 voxels in getting down to the space you would have chosen for just the teapot alone. Reflection rays actually hitting the rest of the stadium would be very sparse, so go ahead and crank up the max subdivision limit.

5) Efficiency issues: The bottleneck in my current algorithm is in finding a coordinate in the next voxel along the ray. This is currently done with rather brute force floating point ray/plane intersections, and takes about 30% to 50% of the time. There are some integer techniques that seem promising, but haven't been tried yet. Eventually it seems that all the coordinate manipulations for walking the octree can be done with integers. This not only works well with point #1 (above), but also would lend itself to a hardware solution more easily. ●

Do you want to receive future copies of the Ray Tracing News?

Check your mailing label for this issue. If your name ends with a /C, it means you're confirmed on the mailing list for future issues. If your name ends with /S or nothing, then you'll have to request to join the mailing list; see the editor's address on Page 1.

Subspaces and Simulated Annealing

by Jim Arvo

This article was originally motivated by the "reflective teapot in a stadium" example devised by Eric Haines as a challenging scene to ray trace. I'll begin by describing how we've encountered and dealt with similar situations. I'll close with some speculation on how a technique called simulated annealing might be brought to bear on problems like this in the future. Most of this work is a result of joint development and countless discussions with Dave Kirk, Olin Lathrop, and John Francis.

One way that we've dealt with situations similar to Eric's teapot example is to use a combination of spatial subdivision and bounding volume techniques. For instance, we commonly mix two or three of the following techniques into a "meta" hierarchy for ray tracing a single environment:

1) Bounding volume hierarchy [Rubin/Whitted,Bouville]

2) Octree [Glassner,Kaplan,Fujimoto]

3) Linear grid subdivision [Fujimoto]

4) Ray Classification [Arvo/Kirk]

We sometimes refer to these as "subspaces". This means a (convex) volume, a collection of objects within that volume, and some technique for intersecting a ray with those objects. This technique is part of an "aggregate object", and all the objects it manages are the "children". Any aggregate object can be the child of any other aggregate object, and appears essentially as a bounding volume and intersection technique to its parent. In other words, it behaves just like a primitive object.

Encapsulating a subspace as just another "object" is very convenient. This is an approach which we originally agreed upon in order to make it possible to "mix and match" our favorite acceleration techniques within the same ray tracer for testing, benchmarking, and development purposes. Several additional benefits emerged from this. For one, aggregate objects also provided a clean way to encapsulate operators, such as "boolean subtract. " More importantly, however, it provided a new way to cope with complex environments.

As an example of the latter benefit I'll describe an amusement park scene which we ray traced and animated. The setting consisted of a number of fairly detailed rides and attractions spread throughout a park, a few trees, a fractal mountain, and two characters who visit several of the rides. We often showed closeups of objects which reflected the rest of the park (a somewhat scaled down version of the teapot reflecting the stadium). There were in the neighborhood of 10,000 primitive objects (not including fractal mountains), which doesn't sound like much anymore, but I think it still represents a fairly challenging scene to ray trace; particularly for animating.

The organization of the scene suggested three very natural levels of detail. A typical example of this is

I) Entire park (a collection of rides, trees, and mountains)

II) Triple-decker Merry-go-round (one of the rides)

III) A character riding a horse (a "detail" of a ride)

Clearly a single linear grid would not do well here because of the scale involved. Very significant collections of primitives would end up clumped into single voxels. Octrees, on the other hand, can deal with this problem but don't enjoy quite the "voxel walking" speed of the linear grid. This suggests a compromise.

Our initial approach was to place a coarse linear grid around the whole park, then another linear grid (or octree) around each ride, and frequently a bounding box hierarchy around small clusters of primitives which would fall entirely with a voxel of even the second-level (usually 16x16x16) linear grid. The low-level bounding box hierarchies were also a way of grouping repeated sub-structures into objects which could be "instanced" without replicating all the geometry. This is similar to the approach taken by Snyder and Barr.

Later, we began to use ray classification at the top level because, for one thing, it did some optimizations on first-generation rays. The other levels of the hierarchy were kept in place for the most part, effectively giving the RC (ray classification) aggregate object a "coarser" view of the world. This drastically cut down the size of the candidate sets it built and allowed it to run well on machines with less than 16 MB of physical memory. Of course, this also "put blinders" on the RC object by not allowing it to

(continued on page 7)

Simulated Annealing

(continued from Page 6)

distinguish between objects inside the "black boxes." This is obviously a space/time trade-off. Being able to nest the subspaces easily provided the flexibility to make trade-offs of this nature.

A small but interesting additional benefit which falls out of nesting subspaces is that it's possible to take better advantage of "sparse" transformations; that is, to transform a ray (or normal) with fewer multiplies and adds by taking advantage of matrices which contain many zeros. Observe that the same trick of transforming the rays into a canonical object space before doing an intersection test (and transforming the normal on the way out) also works for aggregate objects. Though this can mean transforming a ray several times before it even reaches a primitive object, quite often the transforms which are lower in the hierarchy are very simple (e. g. scale and translate). There are cases when a "dense" (i.e. expensive) transform gets the ray into a subspace where most of the objects have "sparse" (i.e. cheap) transforms. If N objects are tested before finding the closest intersection, the job can (occasionally) be done with one dense transform and N sparse ones, instead of N dense transforms. This is particularly true for a complex object which is built largely from scaled and translated primitives and then rotated into some strange final orientation. If it's feasible to make N very small, however, it's often more efficient to just pre-concatenate the transforms and toss the autonomous objects, dense transforms and all, into the parent octree (or whatever). The nesting mechanism is not without its own cost.

Currently, all of the "high level" decisions about which subspaces to place where are made manually and specified in the modeling language. This is much harder to do well than we imagined initially. The trade-offs are very tricky and sometimes counter-intuitive. A general rule of thumb which seems to be of value is to use "adaptive" subspaces (e.g. octree or RC) where there are tight clusters of geometry, and a linear grid if the geometry is fairly uniform. Judicious placement of bounding box hierarchies within an adaptive hierarchy is a real art; it's easy to do more harm than good. On the one hand, you don't want to hinder the effectiveness of the adaptive subspace by creating large clumps of geometry that can't be partitioned. On the other hand, a little a priori knowledge about what's important and where bounding boxes will do a good job can often make a big difference in terms of both time and space (the space part goes quintuple for RC).

Now, the obvious question to ask is "How can this be done automatically?" Something akin to Goldsmith and Salmon's automatic bounding volume generation algorithm may be appropriate. Naturally, in this context, we're talking about a heterogeneous mixture of "volumes," not only differing in shape and surface area, but also in "cost," both in terms of space and time. Think of each subspace as being a function which allows you to intersect a ray with a set of objects with a certain expected (i.e. average) cost. This cost is very dependent upon the spatial arrangement and characteristics of the objects in the set, and each type of subspace provides different trade-offs. Producing an optimal organization of subspaces is then a very nasty combinatorial optimization problem.

An idea which may be of some value is to use "simulated annealing" to find a near-optimal subspace hierarchy. Here "optimality" can be phrased in terms of some scalar-valued objective function which takes relevant factors such as space and time into account. Simulated annealing is a technique for probabilistically exploring the vast solution space of a large combinatorial optimization problem and finding incremental improvements in the objective function without getting stuck in a poor local minimum. It's very closely linked to some ideas in thermodynamics, and was originally motivated by nature's ability to find near-optimal solutions to mind-bogglingly complex optimization problems - like getting all the water molecules in a lake into a near-minimum-energy configuration as the temperature gradually reaches freezing. It's been fairly successful at "solving" NP-hard problems such as the travelling salesman and chip placement (which are practically the same thing).

To apply this technique we think of the objective function as the "energy" of the system (which we want to minimize), and we introduce the notion of gradually dropping "temperature." We produce perturbations in the configuration and then decide whether or not to keep the result. If we were "greedy" and kept only those new configurations which lowered the energy at

(continued on page 8)

Simulated Annealing
(continued from Page 7)

each step, we would descend immediately into a local minimum. This may not buy us much if the initial guess was far from optimal. The simulated annealing approach, on the other hand, accepts a new configuration with probability exp(dE/kT), where dE is the change in energy, and T is the current temperature. This allows more "radical" changes to be explored which initially appear to be counter-productive. As the temperature is lowered, the algorithm becomes more greedy. The all-important strategy for randomization and lowering of temperature is called the "annealing schedule" and is extremely problem dependent.

The applicability of simulated annealing to constructing near-optimal hierarchies is very speculative. It may not be practical at all in this context. One can imagine the annealing being far more costly than the unoptimized ray tracing. There are clearly many details which need to be worked out. For example, one needs to get a handle on the distribution of rays which will be intersected with the environment in order to estimate the efficiency of the various subspaces. Assuming a uniform distribution is probably a good first approximation, but there must be better ways - perhaps through incremental improvements as the scene is ray traced or, better still, between successive frames of an animation.

If this has any chance of working it's going to require an interesting mix of science and "art". The science is in efficiently estimating the effectiveness of a subspace (i. e. predicting the relevant costs) given a collection of objects and a probability density function of rays. The art is in selecting an annealing schedule which will let the possible hierarchies percolate and gradually "freeze" into a near-optimal configuration. Doing this incrementally for an animation is a further twist to simulated annealing to which I've seen no analogies in the literature.

If you're interested in reading more about simulated annealing, there's a very short but interesting description in *Numerical Recipes*. For a more complete treatment see *Optimization by Simulated Annealing*, by S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, in the May 13, 1983 issue of *Science*. Even if this isn't "the way of the future," examining it may lead to some new insights. ●

Top 10 Hit Parade of Computer Graphics Books

by Eric Haines

One of the most important resources I have as a computer graphics programmer is a good set of books, both for education and for reference. However, there are a lot of wonderful books that I learn about years after I could have first used them. Alternately, I will find that books I consider classics are unknown by others. So, I would like to collect a list of recommended reading and reference from you all, to be published later in the year. I would especially like a recommendation for good books on filtering and on analytic geometry. Right now I am reading *Digital Image Processing* by Gonzalez and Wintz and have *A Programmer's Geometry* by Bowyer and Woodwark on order, but am not sure these fit the bill. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling* by Bartels, Beatty, and Barsky looks like a great resource on splines, but I have read only four chapters so far so am leaving it off the list for now.

Without further ado, here are my top ten book recommendations. Most should be well known to you all, and so are listed mostly as a kernel of core books I consider useful. I look forward to your additions!

The Elements of Programming Style, 2nd Edition, Brian W. Kernighan, P.J. Plauger, 168 pages, Bell Telephone Laboratories Inc, 1978.

All programmers should read this book. It is truly an *Elements of Style* for programmers. Examples of bad coding style are taken from other textbooks, corrected, and discussed. Wonderful and pithy.

Fundamentals of Interactive Computer Graphics, James D. Foley, A. Van Dam, 664 pages, Addison-Wesley Inc, 1982.

A classic, covering just about everything once over lightly.

Principles of Interactive Computer Graphics, 2nd Edition, William M. Newman, R.F. Sproull, 541 pages, McGraw-Hill Inc, 1979.

The other classic. It's older (e.g. ray-tracing did not exist at this point), but gives another perspective on various algorithms.

(continued on Page 9)

Top 10 Books
(continued from Page 8)

Mathematical Elements for Computer Graphics, David F. Rogers, J.A. Adams, 39 pages, McGraw-Hill Inc, 1976.

An oldie but goodie, its major thrust is a thorough coverage of 2D and 3D transformations, along with some basics on spline curves and surfaces.

Procedural Elements for Computer Graphics, David F. Rogers, 433 pages, McGraw-Hill Inc, 1985.

For information on how to actually implement a wide variety of graphics algorithms, from Bresenham's line drawer on up through ray-tracing, this is the best book I know. However, for complicated algorithms I would recommend also reading the original papers.

Numerical Recipes, William H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, 818 pages, Cambridge University Press, 1986.

Chock-full of information on numerical algorithms, including code in FORTRAN and PASCAL (no "C", unfortunately). The best part of this book is that they give good advice on what methods are appropriate for different types of problems.

A First Course in Numerical Analysis, 2nd Edition, Anthony Ralston, P. Rabinowitz, 556 pages, McGraw-Hill Inc, 1978.

Tom Duff's recommendation says it best: "This book is *so good* that some colleges refuse to use it as a text because of the difficulty of finding exam questions that are not answered in the book". It covers material in depth which *Numerical Recipes* glosses over.

C: A Reference Manual, Samuel P. Harbison, G.L. Steele Jr., 352 pages, Prentice-Hall Inc, 1984.

A comprehensive and comprehensible manual on "C".

The Mythical Man-Month, Frederick P. Brooks Jr, 195 pages, Addison-Wesley Inc, 1982.

A classic on the pitfalls of managing software projects, especially large ones. A great book for beginning to learn how to schedule resources and make good predictions of when software really is going to be finished.

Programming Pearls, Jon Bentley, 195 pages, Bell Telephone Laboratories Inc, 1986.

Though directed more towards systems and business programmers, there are a lot of clever coding techniques to be learnt from this book. Also, it's just plain fun reading.

As an added bonus, here's one more that I could not resist:

Patterns in Nature, Peter S. Stevens, 240 pages, Little, Brown and Co. Inc, 1974.

The thesis is that simple patterns recur again and again in nature and for good reasons. A quick read with wonderful photographs (my favorite is the comparison of a turtle shell with a collection of bubbles forming a similar shape). Quite a few graphics researchers have used this book for inspiration in simulating natural processes. ●

A Rendering Trick and Puzzle

by Eric Haines

One common trick is to put a light at the eye to do better ambient lighting. Normally if a surface is lit by only ambient light, its shading is pretty crummy. For example, a non-reflective cube totally in shadow will have all of its faces shaded the exact same shade - very unrealistic. The light at the eye gives the cube definition. Note that a light at the eye does not need shadow testing - wherever the eye can see, the light can see, and vice versa.

The puzzle: Actually, I lied. This technique can cause a subtle error. Do you know what shading error the above technique would cause? [hint: assume the Hall model is used for shading].

Efficiency Tricks

by Eric Haines

Efficiency Tricks, by Eric Haines

Given a ray-tracer which has some basic efficiency scheme in use, how can we make it faster? Some of my tricks are below - what are yours?

[HBV stands for Hierarchical Bounding Volumes]

Speed-up #1: [HBV and probably Octree] Keep track of the closest intersection distance. Whenever a primitive (i.e. something that exists - not a bounding volume) is hit, keep its distance as the maximum distance to search. During further intersection testing use this distance to cut short the intersection calculations.

Speed-up #2: [HBV and possibly Octree] When building the ray tree, keep the ray-tree around which was previously built. For each ray-tree node, intersect the object in the old ray tree, then proceed to intersect the new ray tree. By intersecting the old object first you can usually obtain a maximum distance immediately, which can then be used to aid Speed-up #1.

Speed-up #3: When shadow testing, keep the opaque object (if any) which shadowed each light for each ray-tree node. Try these objects immediately during the next shadow testing at that ray-tree node. Odds are that whatever shadowed your last intersection point will shadow again. If the object is hit you can immediately stop testing because the light is not seen.

Speed-up #4: When shadow testing, save transparent objects for later intersection. Only if no opaque object is hit should the transparent objects be tested.

Speed-up #5: Don't calculate the normal for each intersection. Get the normal only after all intersection calculations are done and the closest object for each node is know: after all, each ray can have only one intersection point and one normal. (Saving intermediate results is recommended for some intersection calculations.)

Speed-up #6: [HBV only] When shooting rays from a surface (e.g. reflection, refraction, or shadow rays), get the initial list of objects to intersect from the bounding volume hierarchy. For example, a ray beginning on a sphere must hit the sphere's bounding volume, so include all other objects in this bounding volume in the immediate test list. The bounding volume which is the father of the sphere's bounding volume must also automatically be hit, and its other sons should automatically be added to the test list, and so on up the object tree. Note also that this list can be calculated once for any object, and so could be created and kept around under a least-recently-used storage scheme. ●

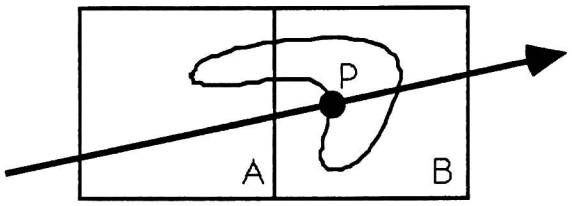
Efficient Ray Tracing in Subdivided Space

by Andrew Glassner

I've been tracing rays in subdivided space for a few years now, and I've come up with a bunch of tricks which are yet to be published. Some of these methods are well known, and I'm aware of simultaneous but independent invention of at least two of these techniques.

Repeated Intersections

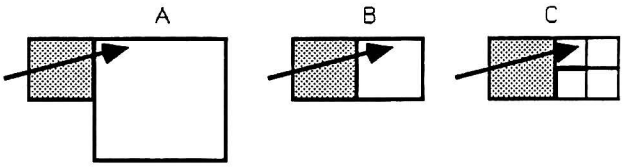
The first issue is easy: how to avoid repeating ray-object intersections when an object straddles several cells. For example, consider the following 2d scenario, where the ray enters box A, misses the boomerang, but then strikes it in box B:



When the ray enters box A, it computes its intersection with the boomerang (point P). The ray then determines that P is outside of A, so the ray moves on to box B. It doesn't make sense to repeat the intersection again. We avoid this needless expense with a "ray intersection cache": we give every ray a unique identification (I use sequential integers starting at 1, since that also gives me a total ray count at the end of the rendering). Every object stores within it two fields, one each for a ray ID and a ray parameter value (the scalar s in the ray equation **R** = **R0** + **R1**s). When we want to intersect a ray and object, the object first checks the ray number it contains against the current ray. If they differ, we calculate the ray-object intersection, store the ray identification and the ray parameter in the object's cache, and return the ray parameter. If the cache ID matches the ray ID, we immeidately return the ray parameter stored in the object, thereby avoiding the intersection. John Amanatides first published this in Eurographics 87.

Inter-Cell Movement

The next issue involves techniques for quickly getting from one cell to the next. Here's another way to approach that issue, which I call "neighbor pointers". Consider the following 2d diagrams, which illustrate a ray leaving a cell (which we'll call Z) through the right-side wall:



Here we have 3 three possible situations that can arise when we leave the shaded cell and pass into the next. In case A, we pass into a larger cell; in case B, we pass into a cell of equal size ; and in case C, we pass into a smaller cell. Note that for cases A and B, it doesn't matter where we actually pass through the wall when we exit the shaded cell: we will always end up in the same cell. Case C is not quite so simple, since the cell we enter next is dependent on the position of the ray on the wall when it passes out of the first cell. What we know for certain, though, is that the next cell will be a child of a cell the same size as the cell we're leaving.

To optimize traversal between cells, we can pre-process the entire database. Each wall of each cell is given a "neighbor pointer". This is the address of the data structure for the appropriate cell on the other side of that wall. For cases A and B, we store the single cell that shares that wall's boundary. For case C, we store the cell of the same size as the cell we're processing. The general rule is that the neighbor of cell Z across wall W is the smallest cell that shares all of W as seen by Z.

Now let's consider actually tracing a ray. We determine that there's no appropriate intersection in this box, and we must move on. We find the nearest wall hit by the ray, and dereference the neighbor pointer for that wall. We examine the subdivided flag in that cell description. In cases A and B, the flag is off, and we're done: we have the cell description (with its children, boundaries, and so on) immediately, with no hashing or other expensive work. In case C, the flag will be on. Then I compute a point that is

(contintued on page 12)

Efficient Ray Tracing

(continued from Page 11)

guaranteed to be in the next cell, and descend the octree from the cell pointed to by the neighbor pointer. We stop descending the tree when we hit a child leaf.

In effect, cases A and B are instantaneous movements up and across the tree, repectively. Case C requires a descent. An alternative way to handle case C is to build a quadtree on the wall of the cell, which mimics the octree structure on the other side of that wall. We descend the quadtree based on the ray/wall intersection, and then use the neighbor pointer held by that quadtree cell. I don't bother with this, since it takes up some memory to store that quadtree, and it only duplicates the octree structure.

This technique of pre-processed wall pointers was independently developed by Arun Netravali and co-workers at Bell Labs.

Subdivision Techniques

Based on what criteria do we subdivide? I don't know of a single good answer.

Rather than manage several criteria for each cell, I apply different criteria in sequence. For example, one sequence that has worked well for me is to subdivide on an object-count criterion until every node either meets the criterion or is split. I then process the leaves according to a density criterion: if the ratio of the volume occupied by the objects to the volume of the cell is too low, I subdivide. Obviously the object count and density threshold are numbers that have to be tweaked by hand for optimum performance. I use 3 objects and a density of .3 most of the time, and that seems to work well. Pathological cases (or very unlucky orientations of the objects relative to the octree) can require some intervention.

Incidentally, the notion of "object count" should be modified a little bit in the case of polygons and curved surfaces tied to polygonal control patches. In these cases, I use a vertex count instead. To see why, imagine a many-faceted diamond, where maybe 30 faces could share a single vertex. That vertex will land in some cell, and no matter how much we subdivide that cell we're still going to have lots of faces (some could get eliminated if/when the vertex is right on a cell wall, but that might take many generations of subdivision). So I accept the many faces and move on; the density check that comes later will tend to make the box around this vertex quite small.

Shorter Trees

Eric's "teapot in a football stadium" is a good demonstration case for the power of what I call "single-child replacement". In this technique, we examine the children produced by subdividing a node. If all of the children are empty except for one child, then we replace the parent node by its single, non-empty child. A bit of clever programming lets us manage this transparently. So the many smaller, useless generations of cells that would be generated between the stadium and the teapot disappear as we descend to the tiny teapot cells. ●

Production Notes

The main text of this newsletter was set in 10-point Bookman type. The top and bottom banners are 12-point Helvetica Bold, and article titles and bylines are 18 and 14-point Avant Garde, typeset at 300 dpi.

Articles from authors at remote sites were sent to the editor in Chapel Hill via electronic mail, using the UUCP network and the Arpanet. Upon arrival at UNC the articles were saved as ASCII files on our mail VAX-11/780. Articles were transferred from the VAX to a Macintosh using NCSA Telnet 2.0. Initial formatting was prepared with MacWrite 4.6. When layout was ready to begin, each article was read into Word 3.01 using a custom format designed for the newsletter. After some further editing each article was saved in this new format.

Illustrations and the flag logo were executed in MacDraw 1.9.5 by the editor.

The layout was executed using PageMaker 2.0a on a Macintosh II with a SuperMac Technology large color screen. The final pages were printed on a LaserWriter Plus and then photo-reproduced.

The mailing list is maintained with Word 3.01, which also printed the labels.

Design and Layout by Andrew Glassner