The Ray Tracing News

Volume 1, Number 1 "Light Makes Right" September 1987

Contents

Letters to the editor will be considered submitted for publication unless otherwise requested. Please address all correspondence, including subscription requests, to the editor at the addresses given below. Material may be submitted on hardcopy, via electronic mail, or on a Macintosh 3.5 inch diskette in either MacWrite or text-only formats. All submissions will normally be acknowledged via electronic or physical mail. Physical mail to the Netherlands should be clearly marked on the envelope "Scientific Technical Matter – No Commercial Value" to avoid delays in customs.

Editor's address: Andrew Glassner
(to 20 Dec 87) Delft University of Technology
Faculty of Mathematics and Informatics
Julianalaan 132
2628 BL Delft
The Netherlands
mcvax!dutrun!frits
phone (015) 78-2528

(after 20 Dec 87) UNC-Chapel Hill
Department of Computer Science
Sitterson Hall
Chapel Hill, NC 27514
mcnc!unc!glassner

Newsletter Announcement and Statement of Purpose

This newsletter is intended to be an informal arena for discussions and news relevant to the field of ray tracing, particularly in computer graphics.

At Siggraph '87 a number of folks gathered together one evening just to discuss ray tracing and related issues (this was a sequel to the successful ray tracer's dinner during Siggraph '86). Several people at that discussion expressed a desire to keep in contact, and my original plan was just to distribute a list of names and addresses to everyone who had been there.

But before returning home from Siggraph to prepare that list I spent a couple of weeks hiking in the Rockies, and an alternative idea began to hatch. I realized that lots of the information we share when face-to-face gets repeated when in different groups. And lots of ideas are discussed that are useful and clever, but not sufficiently deep or novel for formal publication. And many practical techniques are shared; some are published but still unknown, others are new inventions.

I felt that we could use a year-round forum, similar to the yearly meetings at Siggraph, where we could discuss issues that are interesting to us all.

In response to these feelings, The Ray Tracing News has been created to publish information of the following types:

- Notices of events of interest to the ray tracing community.
- Timely technical information of a useful or practical nature.
- Articles and letters discussing or debating interesting topics.
- Information to help researchers share ray tracing code and databases.
- Algorithms, solutions, or approaches to interesting problems.

(continued on Page 2)

A Summary of the Ray Tracing Roundtable at Siggraph '87

Andrew Glassner

The 1987 Siggraph Ray Tracing Roundtable was an informal gathering of about 20 ray tracing researchers early Thursday evening. The evening began with a discussion of how to measure the performance of various ray tracing algorithms. Although many interesting ideas were suggested, the group never seemed to even come near a consensus. It seemed as though ray tracing algorithms must (at least for the time being) be compared when rendering a particular scene, and even then one must know a good deal about the geometry and physics of that particular database. Most folks agreed that "standard databases" and "standard images" were a good first step in this direction, giving everyone a yardstick against which to measure.

We then discussed the current collection of what objects can be ray traced. It was no surprise that most objects in ray traced images have well-defined surfaces, although there are a few notable exceptions, such as density volumes. We agreed it would be nice to be able to ray trace fuzzier surfaces with less sharply defined boundaries.

The subject of caustics drew a range of opinions. Some believed it to be virtually a solved problem, others thought that very little was known about the effect and how to simulate it. We discussed the approaches advocated in print, as well as some ideas that had not yet been published. It seemed that forward ray tracing of some form would be necessary, although how that could be efficiently achieved was not obvious.

We discussed the differences in point-of-view between those people working on algorithms for hardware implementation and those working in software. Clearly ray tracing is a very simple algorithm to place onto most parallel machines with good results. But perhaps the algorithms can be structured in such a way that new, custom hardware can make a dramatic contribution. Recent successes of machines with a very large number of simple processors have made this approach particularly attractive. On the other hand, the flexibility of software implementations argues that new research is still practical on uniprocessors.

The intriguing comparisons between ray tracing and radiosity were then discussed, but most folks demurred stating an opinion on this topic until they had read and considered the last paper of the conference, which presented a single algorithm combining the two techniques.

The last discussion involved how an algorithm can be strongly influenced by the nature of the final images; are they to be used as frames in an animation or as still images? Some positive results of work into frame coherence for animation were mentioned and discussed.

We closed with a general feeling that our time had been well spent, and a desire to do it again next year. Of course, small informal discussion groups formed spontaneously throughout the conference, both before and after the roundtable, to discuss these topics, new research, and crazy ideas. ●

Eric Haines suggested an interesting quesiton recently. He wants to compute ray intersections with bicubic patches. He'd also like to support rational curves, non-uniform knots, and trimming curves. Some folks intersect patches by turning them into a mesh of polygons before rendering. Others subdivide on the fly for a particular intersection. Do you ray-trace patches? Let us know how you do it. Share your observations and experiences in a short note or article.

Newsletter Announcement

(continued from Page 1)

- Reviews or discussions of commercial products.
- Fun or humorous material.
- Other information relevant to ray tracing.

Submissions may be made by physical mail, electronic mail, or on a Macintosh disk, as discussed in the fine print on page 1. I will endeavor to publish all material as received, without review, as long as it remains a manageable quantity and the quality is high.

Please feel free to submit material that you think is relevant to the ray tracing community, even if it's not polished results. This is meant to be our own forum, where we can address and discuss issues that we feel are relevant. These issues may concern ray tracing itself, or the technique in wider contexts, such as image synthesis or computer graphics as a whole, or even ray tracing in other fields.

I welcome your comments and submissions.
-Andrew ●

Efficient Boolean Evaluation of CSG Models for Ray Tracing

Andrew Glassner

There are many ways to handle Constructive Solid Geometry in a ray tracing environment. Some CSG techniques for ray tracing are more efficient than others, but most of them are surprisingly easy to implement.

In this article we will restrict our discussion of CSG to four operators (three binary, one unary) on primitive solids. Each operator has a formal mathematical symbol and semantics, but they also have ASCII nicknames, which we will use here. The binary operators are union (+), intersection (&), and difference (–):

$A+B = A \vee B$; $A\&B = A \wedge B$; $A-B = A \wedge \sim B$

The unary operator is not (~), which simply inverts its argument. We build complex solids by creating a binary tree, consisting of primitive solids at the leaves and CSG operators at the nodes. To intersect a composite CSG object wefind the intersections with the primitive solids and determine which of these intersections (if any) is also an intersection with the composite CSG object.

A common theme in the CSG algorithms we'll look at here is that they ask when a ray first enters (or finally leaves) the composite CSG object. This is equivalent to looking for a change in status of the root node. Each time we find an intersection with a primitive, we are either entering or leaving that solid. So we can say whether, just after the most recent intersection, we are "inside" or "outside" each primitve. We can thus also decide if we're "inside" or "outside" of each node in the tree. When the root changes from outside to inside, then the current intersection is a valid entry to the composite CSG object. When the root changes from inside to outside, then we're leaving the CSG object. Note that even if all primtive solids are convex, the composite solid built from them in a CSG tree is usually not convex, so it is possible to enter and leave the composite solid several times.

Perhaps the most straightforward CSG algorithm is the method of Roth [Roth82]. This technique finds all intersections where the ray enters and leaves each primitive object along its path. The intersections are sorted along the ray path and Boolean operations are used to find the sections of the ray where it is within the composite solid; the first endpoint of such a span (measured from the origin of the ray) is the first valid ray-object intersection with the composite CSG object. A major drawback is that the approach as presented requires all intersections between the ray and the primitive solids in the database. Unfortunately, once the first valid intersection is found the others are useless, and we all know that we want to avoid computing as many of these intersections as we can.

Another way to go is to collect the intersections along the ray path as they occur. Each intersection is tested against the composite solid until the first valid intersection with the complete CSG object is found. So we're effectively watching the progress of the head of the ray as it passes through the database. We're looking for events where the head of the ray enters or leaves the composite CSG object. There are at least two general ways to test the primtive intersections as they occur, and they refer to the order in which we process the CSG tree when we want to classify an intersection: top-down and bottom-up.

In the top-down approach you start at the root node of the tree and evaluate its left and right children. If as a result of this evaluation the root node changes state, then the intersection event that caused you to re-evaluate the tree is a valid entry or exit with the composite object. Top-down evaluation is recursive; to find the state of a child you must evaluate it. Recursion stops at the leaf nodes, where the head of the ray is either inside or outside the primitive solid at that leaf. There are a few disadvantages associated with top-down CSG evaluation. For example, you need to know a lot about the inside/outside status of the tree nodes. Some children need not be evaluated if you're clever (e.g. if an intersection node determines one child is "outside," it need not test the other child, since it's irrelevant to the status of the node), but there's still a lot of testing to be done. And at each node you have two paths to descend; maybe one gives you a quick classification of the node status, and the other doesn't; I know of no algorithm that tells you, in general, which is the more fruitful path to descend first.

A way I like better is the bottom-up approach, presented by Bronsvoort [Bronsvoort86]. In the bottom-up approach we attach a unique copy of the CSG tree with every ray. Each CSG node contains a pointer to its parent, and a Boolean bit which indicates whether the head of the ray is currently inside

(continued on Page 4)

Efficient CSG

(continued from Page 3)

or outside the subtree rooted at that node. For leaf nodes this bit indicates whether the head of the ray is inside or outside that node's primitive solid; for internal nodes the bit describes whether the head of the ray is inside or outside the composite CSG tree rooted at that node.

To summarize the technique, imagine that we have a primary ray which begins outside of all objects; thus every node in the CSG tree associated with this ray is marked "outside". We use standard techniques (with a slight change described below) to find the first intersection of this ray with the objects in front of it. Let's say we find that we've hit some object. We locate the leaf node containing that object, toggle its Boolean inside/outside flag from "outside" to "inside", and then signal that node's parent. The parent node looks at its children and determines whether its status has just changed; that is, it determines whether the ray has either entered or left the composite CSG object for which it is the root. If the node finds that its state has not changed as a result of this intersection, then the bottom-up tree propagation stops right there. We accept that intersection as invalid as far as the composite CSG solid is concerned, and we proceed to find the next intersection, for which we will repeat the whole process.

Alternatively, if the node determines that its status has changed as a result of this intersection, it flips its status bit and signals its parent. If the node has no parent then it is the root of the complete CSG tree, and the ray has made a valid entry or exit with a complete CSG object, depending on the inside/outside status of the root node.

A modification to the intersection algorithm is what enables us to find the "next intersection" along a ray's path. It is not difficult to enhance most ray tracing programs to get intersections along a given ray beyond the first. If the program always finds all intersections between the ray and the database, then a sorted list of the intersections may be stored with the ray. When successive intersections are required, they are simply taken in order along the list.

If the program uses some form of acceleration, then the algorithm can usually be easily modified to give successive intersections. Space subdivision algorithms can retain pointers to the last spatial region or cell entered and the latest object intersected in that region, and the value of the ray parameter at that intersection. When another intersection is required ray/object testing begins again in that cell, only accepting intersections beyond the one stored. Bounding volume algorithms can be enhanced in a similar way, keeping the stack of bounding volumes around, with enough information to provide re-entry to the processing code to find the next intersection.

When a given node is signalled during bottom-up evaluation, it must determine if its inside/outside status has changed as a result of this intersection. Leaf nodes (which point to objects) are easy; when signalled they always toggle their bit and signal their parent.

Internal CSG operator nodes are slightly harder. There are several ways to go about finding whether one of these nodes should change state. Perhaps the simplest approach is to evaluate the operator with the states of its current children, and then compare the new state with the. old state and proceed as above based on whether the old and new states are the same or different.

Another approach for internal nodes is to compute a Boolean variable from on the current children and the old state; this variable is true when the new state is different and otherwise false. The formula for this Boolean is slightly different for each of the three operators. The diagram on page 5 gives a (rather lengthy) derivation of these formulae.

To summarize the working of the algorithm for a ray originating outside of all objects, we initially set all nodes to "outside". When we hit an object, we signal its node. The leaf node associated with that object toggles its inside/outside bit and signals its parent. The parent uses one of the techniques described above to determine if its status has just changed. If not then the node does nothing and the intersection is deemed ineffective; the ray tracer then looks for the next intersection. Alternatively, if the status has changed, then the node flips its status bit and signals its parent, where the process repeats. If the root node ever flips its status bit, then the intersection that caused the change is a valid intersection with the composite CSG object. If after flipping its status the root is "in", then we have entered the composite object; otherwise we have left it.

Secondary rays work the same way except that their CSG tree is initialized not to "all outside", but to the status of their parent ray's CSG tree at the time of intersection. Before the

(continued on Page 6)

Derivation of Boolean CSG State Change Formulae

S	L	R	S'	T	$\underline{L}$	**L**	$\underline{R}$	**R**	V	F
0	0	0	0	0	1	0	0	1	1	0
0	0	1	0	0	0	1	0	1	1	0
0	1	0	1	1	0	1	0	1	1	1
0	1	1	0	0	0	1	1	0	1	0
1	0	0	0	1	1	1	0	0	1	1
1	0	1	0	1	0	0	0	0	0	~
1	1	0	1	0	0	0	0	0	0	~
1	1	1	0	1	0	0	1	1	1	1

	S		$\overline{S}$	
R	1	~	0	0
$\overline{R}$	~	1	0	1
	L	$\overline{L}$		L

- $S \vee (L \wedge \overline{R})$

L – R

S	L	R	S'	T	$\underline{L}$	**L**	$\underline{R}$	**R**	V	F
0	0	0	0	0	1	0	1	0	0	~
0	0	1	1	1	1	0	0	1	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	1	1	1	0	1	0	0	~
1	0	0	0	1	1	1	1	1	1	1
1	0	1	1	0	1	1	0	0	1	0
1	1	0	1	0	0	0	1	1	1	0
1	1	1	1	0	1	1	1	1	1	0

	S		$\overline{S}$	
R	0	0	1	~
$\overline{R}$	0	1	~	1
	L	$\overline{L}$		L

- $\overline{S} \vee (\overline{L} \wedge \overline{R})$

L + R

S	L	R	S'	T	$\underline{L}$	**L**	$\underline{R}$	**R**	V	F
0	0	0	0	0	0	1	0	1	1	0
0	0	1	0	0	1	0	0	1	1	0
0	1	0	0	0	0	1	1	0	1	0
0	1	1	1	1	0	1	0	1	1	1
1	0	0	0	1	0	0	0	0	0	~
1	0	1	0	1	1	1	0	0	1	1
1	1	0	0	1	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0	0	~

	S		$\overline{S}$	
R	~	1	0	1
$\overline{R}$	1	~	0	0
	L	$\overline{L}$		L

- $S \vee (L \wedge R)$

L & R

One way to determine if a signalled node needs to change state is to evaluate a Boolean expression involving the current state and the current children. If this Boolean is true, then the node should toggle to a new state and signal its parent. If the Boolean is false, then the node does not change state and we must look for additional intersections. Each of the three CSG operators has its own expression to evaluate, although they are all similar in form. The accompanying tables derive the correct expression for each operator.

We begin by listing all possible configurations of the current state S and the current children L and R. We then compute S', the new state based on L and R for the appropriate operator (to simplify the discussion, we will refer to a generic operator Δ, which may be either +, –, or &); thus $S' \approx L \,\Delta\, R$. We then compute the toggle variable T, which indicates whether the new and old states are different; $T \approx S \neq S'$. When T is true then we must toggle. We could stop here and find a Boolean expression for T, but some additional analysis will let us derive a simpler expression.

The analysis consists of finding those configurations that are impossible for our algorithm. For example, a union node may never find itself responding to a signal when its current state is 0, but both children are 1. This is because only one of the children may have just toggled; thus at least one child had to be 1 before we were signalled, so this node had to have a current state of 1. To avoid making such logical arguments for all configurations for all 3 operators, we derive some new variables which will ultimately tell us which configurations are impossible. It is by ignoring these impossible configurations that we are able to find a simpler expression for the toggle variable.

We first ask what state we would be in if the left node had just toggled. Then the previous value of the left child is not-L (written ~L), and the previous state would be $\underline{L} \approx {\sim}L \,\Delta\, R$. If indeed L had just toggled, then our current state would be the value of $\underline{L}$; we compute $\mathbf{L} \approx S = \underline{L}$, which is true if our current state is consistent with the assumption that the left child just changed. We now ask the same questions for the right child, computing $\underline{R} \approx L \,\Delta\, {\sim}R$ and $\mathbf{R} \approx S = \underline{R}$. If neither **L** nor **R** is true, then the current configuration could not have come from either child just toggling; thus this is an impossible configuration. We compute a "valid" variable $V \approx \mathbf{L} \vee \mathbf{R}$. We can now write down a new column F which has the value of T where V is true, but has "don't-care" (written ~) where V is false. After all, it doesn't matter what value we compute for T in configurations that can't occur.

So F is the same as T in all configurations that are possible, and "don't-care" otherwise. We can write the Karnaugh map for F and derive a simple Boolean expression based on S, L, and R, which gives us the correct value of T in all possible configurations. So when a node is signalled, it computes the appropriate expression for F. If F is true, the node must toggle its current state and signal its parent. If F is false, then processing stops and we go looking for more intersections.

A (typographically dense) nugget of C code to implement this algorithm might look like this:

```
signal_version1(node) CSGnode *node; { Boolean F;  /* compute Boolean */
  switch(node->operator) {
    case '-': F = ( node->S) || (( node->lChild->S) && (!node->rChild->S));  break;
    case '+': F = (!node->S) || ((!node->lChild->S) && (!node->rChild->S));  break;
    case '&': F = ( node->S) || (( node->lChild->S) && ( node->rChild->S));  break;
  }
  if (F) { node->S = !node->S;
           if (node->parent != NULL) signal(node->parent);
                                else return(VALID_INTERSECTION);
  } else return(INVALID_INTERSECTION);
}
```

You might want to compare this with a very similar nugget for the alternative approach of computing the new state and then comparing it to the old:

```
signal_version2(node) CSGnode *node; {  Boolean S; /* compute new state */
  switch(node->operator) {
    case '-': S = (( node->lChild->S) && (!node->rChild->S));  break;
    case '+': S = (( node->lChild->S) || ( node->rChild->S));  break;
    case '&': S = (( node->lChild->S) && ( node->rChild->S));  break;
  }
  if (S != node->S) { node->S = !node->S;
                      if (node->parent != NULL) signal(node->parent);
                                           else return(VALID_INTERSECTION);
  } else return(INVALID_INTERSECTION);
}
```

Efficient CSG

(continued from Page 4)

ray continues, though, it might need a correction to its tree. Imagine that our primary ray has just hit the surface of a sphere from the outside; the CSG status of the sphere is then toggled from "outside" to "inside". Now imagine that reflected and transmitted rays are generated at this point. The transmitted ray indeed continues into the body of the sphere, so its CSG status is "inside" the sphere, just as it was initialized by the primary ray. But the reflected ray never enters the sphere; its status should be "outside" the sphere. To perform this correction I first assume that the surface at the point of intersection is locally flat, or at least not extremely curved (the idea is that any refracted ray will be inside, any reflected ray outside). I then compute the dot product of the direction vector of the secondary ray with the surface normal; if the result is greater than zero then this ray has "turned around" with respect to its parent and this object, so I signal the leaf node associated with that object (which propagates the information up the tree by signalling its parent node). This works for all secondary rays, including shadow feelers.

Eye rays must also be initialized correctly; they might start inside of an object. You can correctly initialize an eye ray by creating a special auxiliary ray designed just to find this information, just like how we create shadow rays to determine illumination. Create a ray that begins at the eye and point it in any direction; initialize the CSG status of this ray to all "outside". Now trace the ray until it leaves the database. When you hit a primitive object with this ray, invert the inside/outside bit for that node, but don't bother having it signal its parent. When you finally leave the database, the status of the CSG tree for that ray is the correct initial status for all rays beginning at that fixed eye point.

The bottom-up CSG algorithm can be implemented quite simply and efficiently; only maintaining the CSG status trees requires any really new code (adding enough state to the intersection routines to allow them to avoid repeating work for successive intersections is theoretically easy, but the implementation could be of any difficulty, depending on your system).

Another advantage of the bottom-up approach is that you have the freedom to model with many small CSG trees in your database, and you'll only need to evaluate the nodes relevant to a particular intersection. In this situation, toggling any root node indicates a valid CSG intersection.

Appendix I

It's important that the state of the CSG tree of any secondary ray (shadow, reflection, or refraction) be correctly initialized from its parent. Since most of the time we generate more than one secondary ray at a surface, we need some efficient way to pass the status of the parent's CSG tree to each child.

Somewhere in the ray-spawning code there's probably some kind of stacking mechanism. This may be explicit (maintaining your own stack) or implicit (calling a function or routine with a new ray description as a parameter, and letting the system keep the stack in the calling frame). Either way, it would be inefficient to create a new copy of the entire CSG tree for each secondary ray; the CSG tree in a complex environment might have thousands of nodes!

Of course, we can always trade space and time. We could encode the status of the entire CSG tree with some identifier. An extreme example would be to use Gödel numbering of the status and node number, converting the entire tree into some large integer. We could associate this identifier with the ray and reconstruct the tree when necessary. This is probably too difficult and time consuming for practical use.

The other extreme is to store the entire tree with each ray. I'd like to avoid that, because memory consumption will go up very quickly, since a ray tree typically has many nodes.

A compromise that has worked acceptably for me is to create a single extra copy of the entire CSG tree in global memory. Each time we generate a secondary ray we associate with that ray a list of all primitive solids which are classified "inside" in the parent's CSG tree. When we need the CSG tree for a given ray, we take the global tree and initialize all the leaf nodes to "outside". We then run through the list of primitives associated with the ray; the leaf node corresponding to each of these primitives is adjusted to be "inside". We then pass through each of the interior nodes of the tree one level at a time, from bottom to top, evaluating the status of each node based on its children. In this process we don't bother with signalling the parents of nodes that change their state, since we're not evaluating an intersection;

(continued on Page 7)

Efficient CSG

(continued from Page 6)

we're just adjusting the interior nodes to be consistent with the leaf nodes. When this step is complete, the global tree now holds the correct status for starting out that secondary ray.

As with most algorithms, a bit of hacking can make this scheme go a bit quicker. For example, instead of building and storing an identical list with every secondary ray, create the list once, associate it with the primary ray (the ray which caused the new rays to be spawned), and let secondary rays point to the list in the primary. When we finally are finished with the primary ray we can release its list to free memory.

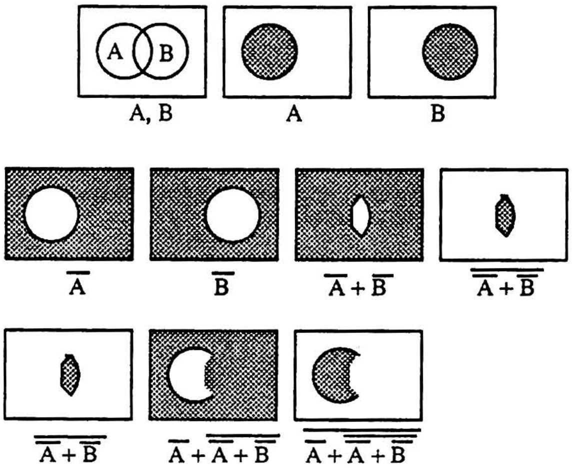
In the limit, the "inside" list can contain every leaf node in the tree. We can ameliorate this problem by keeping count of how many elements are in the "inside" list. When we have more than half of the leaf nodes, we flip a bit at the head of the list and instead store the "outside" nodes. Thus the list will never contain more than half of the leaf nodes in the CSG tree.

Appendix II

It is well known that ~ and ∧ suffice for writing all Boolean expressions. It was pointed out to me by Frits Post that we may also write the CSG operators in terms of 2 CSG primitives. If we choose (~,+) then

A–B = ~(~A + ~(~A + ~B))
A&B = ~(~A + ~B)

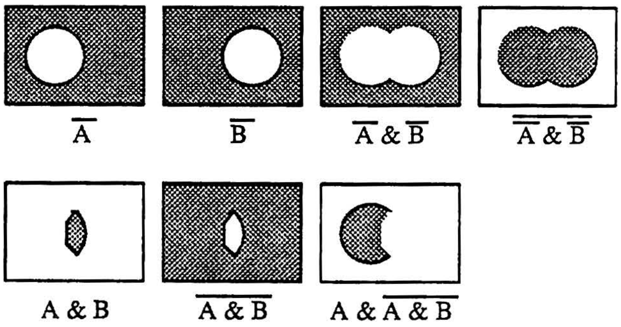
Here are some set diagrams demonstrating these equivalences:


A
B
A, B
A
B
$\overline{A}$
$\overline{B}$
$\overline{A}+\overline{B}$
$\overline{\overline{A}+\overline{B}}$
$\overline{\overline{A}+\overline{B}}$
$\overline{A}+\overline{\overline{A}+\overline{B}}$
$\overline{\overline{A}+\overline{\overline{A}+\overline{B}}}$

or if we choose (~, &) as primitives, then

A+B = ~(~A & ~B)
A–B = A & ~(A & B)

which we can diagram:



In fact these equivalences are direct applications of the class-theoretical expression of De Morgan's law:

~(A + B) = ~A & ~B
~(A & B) = ~A + ~B

Expressing the entire tree in terms of (~, &) is useful because it can sometimes be detected when A&B is empty. In this case the tree simplifies, so that A&B nodes can be ignored, and A–B nodes turn into just A. The field of tree rewriting and simplification is still growing; for example see [Goldfeather86], [Verhoeve87], and [Jansen87]

Another interesting topic appropriate to CSG ray tracing I first heard from Wim Bronsvoort. Let's say that we have a union of two spheres, one of which is somewhat transparent and the other opaque. What should be the result of subtracting this partly transparent sphere from the solid one? Consider a point of view looking towards the solid sphere through the translucent one. Do you see the surface of the solid sphere where it intersects the other? As far as I am aware, the semantics of CSG for non-homogeneous objects have not been well defined. ●

References

[Roth82] Roth, S.D., "Ray Casting for Modelling Solids," *Computer Graphics and Image Processing*, 18 (2), February 1982, pp. 109-144

[Bronsvoort86] Bronsvoort, W.F., "Techniques for Reducing Boolean Evaluation Time in CSG Scan-line Algorithms," *Computer-Aided Design*, 18 (10), December 1986, pp. 533-538

[Goldfeather86] Goldfeather, J., J.P.M. Hultquist, H. Fuchs, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System," *Computer Graphics* 20 (4), August 1986, pp. 107-116

[Jansen87] Jansen, F.W., "Solid Modelling with Faceted Primitives, Doctoral Dissertation, Department of Industrial Engineering, Delft University Press, 1987

[Verhoeve87] Verhoeve, P., "Extended Octtrees and Their Application to Boolean Shape Operations on Polygonal Objects," Engineer's Thesis, Delft University of Technology, January 1987

A Proposal for Standard Graphics Environments

Eric Haines
3D/Eye, Inc

One concern of the computer graphics community has been the efficiency of rendering algorithms. In fields such as ray tracing, researchers continue to explore which is the fastest way to find the closest intersection point for a ray and a set of primitives. The problem faced by these and other people involved in computer graphics is a lack of standards.

In the hardware field a metric of polygons per second is used. In ray tracing, however, the rendering of a single primitive, such as a polygon, can be affected by the other primitives in the environment. For example, another primitive could cast a shadow or be seen in reflection from the primitive being rendered. This has led to timing comparisons based on the time for calculating ray intersections, instead of a primitives per second rate. One of the problems with trying to compare ray intersection times is that there are almost no standard test environments. One researcher will ray trace a car; another, a tree. The question arises, "How many trees to the Camaro?"

My proposal is that we should all be using the same environments. I originally heard of this idea from Don Greenberg while I was in Cornell's Program of Computer Graphics. He and Ed Catmull had once discussed producing some environments which would be used as standards for testing rendering algorithms. A few years later, Tim Kay presented a paper on efficient ray tracing at Siggraph '86. He offered his database descriptions to any researcher who wanted to use them. Discussions with him and other researchers led me to create a number of scenes for testing ray tracing algorithms.

The databases are fairly familiar and "standard" to the graphics community. The scenes are generated by the "Standard Procedural Database" package, or SPD for short. Each database is generated by a program written in C. The output of the program is in text, with information about the view, lighting conditions, and primitives being output in a simple format. Presently polygons, polygonal patches (polygons with a different surface normal at each vertex), spheres, cylinders, and cones are supported. The researcher has to write a program to translate these simple output data into the format needed by the algorithm or hardware being tested.

The SPD package is in the public domain and can be accessed in a number of ways. Netlib is distributing the package for free. For those with access to the Arpanet, write to "netlib@anl-mcs.arpa". If electronic mail on the Unix UUCP network is available, write to "research!netlib". In either case, send the one line message, "Send Haines from graphics."

A few extra words are in order on netlib. This library is mostly code for numerical analysis people, but there are some great hunks of software and data useful for graphics. Send the message "send index" for a general index. The library I like personally is Polyhedra, which contains 142 descriptions of polyhedra (vertices, edges, faces, and more). Access this by typing "send index from polyhedra". If you know of any good public domain graphic's software or databases to post to netlib, please write them ("send index" will give you names and addresses).

An early, incomplete version of the SPD package was printed as an appendix in the notes for the "Introduction to Ray Tracing" course given at Siggraph '87. For the IBM PC, the package is available on a 360K 5-1/4" floppy disk. Send a stamped, self-addressed disk mailer and blank disk to: Ed Orcutt, Computer Science & Electrical Engineering, University of Nevada - Las Vegas, 4505 Maryland Parkway, Las Vegas, NV 89154. If none of these media are available to you, send $4 for the latest printed version from me: Eric Haines, 3D/Eye Inc., 410 E. Upland Rd, Ithaca NY 14850.

Presently this package is simply a proposal. Your feedback is needed on a number of questions, such as what constitutes an average scene for your applications, what primitives you use, and your opinions on the package in general. Timings and statistics for different ray tracing algorithms are also most welcome. ●

(*Editor's note* : these images will be published in IEEE CG&A, probably in the November 1987 issue).

If you are not on the mailing list printed in this issue...

...then this copy was mailed to you because I thought you might be interested. If you would like to receive future issues of the Ray Tracing News send your name and address to me at the address given in the fine print on page 1. Please also include an electronic mail address, if you have one.

A Small Catalog of Reflectance Spectra

Andrew Glassner

Over the years I have collected a bunch of useful spectra. Most have come from published literature, others I made myself just by playing around. For historical reasons I keep each spectrum in 14 equal samples from 380 to 770 nanometers. Here are the spectra, arranged into two columns each.

380	590
410	620
440	650
470	680
500	710
530	740
560	770

Wavelengths

0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000
0.000	0.000

Blackbody (M)

0.000	1.000
0.000	1.000
0.000	1.000
0.000	0.000
0.000	0.000
0.000	0.000
0.050	0.000

Brick Red (I)

0.110	0.453
0.132	0.490
0.160	0.524
0.195	0.550
0.240	0.565
0.319	0.565
0.393	0.565

Bronze (M)

0.065	0.066
0.065	0.067
0.065	0.067
0.066	0.068
0.066	0.070
0.066	0.070
0.066	0.070

Carbon (M)

0.169	0.411
0.226	0.414
0.283	0.415
0.336	0.416
0.390	0.415
0.398	0.413
0.405	0.410

China Clay (M)

0.04	0.51
0.07	0.61
0.12	0.70
0.18	0.78
0.25	0.86
0.33	0.94
0.42	1.00

CIE A (E)

0.22	0.95
0.41	0.96
0.78	1.00
0.89	1.00
0.91	0.93
0.89	0.84
0.99	0.82

CIE B (E)

0.27	0.70
0.65	0.71
0.98	0.71
1.00	0.68
0.91	0.58
0.79	0.50
0.85	0.47

CIE C (E)

0.070	0.552
0.088	0.670
0.118	0.711
0.156	0.740
0.205	0.755
0.250	0.755
0.335	0.755

Copper (M)

0.376	0.848
0.378	0.868
0.372	0.880
0.382	0.898
0.400	0.910
0.640	0.910
0.812	0.910

Gold (M)

0.00	0.000
0.00	0.000
0.00	0.000
0.00	0.000
0.50	0.000
1.00	0.000
0.50	0.000

Leaf Green (I)

The left hand column contains amplitudes from 380-560 nanometers. Ther right hand column contains amplitudes from 590-770 nanometers. Next to the name of each spectrum I have indicated its source:

(E) = "An Introduction to Color," by R. Evans, published by John Wiley and Sons, 1948
(I) = Imaginary color
(M) = appendix to "Colorimetry and Computer Graphics," by Gary Meyer and Donald Greenberg, unpublished (I believe the data in that appendix was collected by Rob Cook)

0.105	0.127
0.078	0.130
0.112	0.133
0.114	0.135
0.117	0.136
0.120	0.137
0.124	0.133

Lunar Dust (M)

0.081	0.025
0.087	0.053
0.092	0.082
0.094	0.111
0.097	0.124
0.067	0.126
0.037	0.123

Lunar Rock (M)

0.310	0.515
0.344	0.530
0.380	0.545
0.415	0.558
0.445	0.565
0.471	0.565
0.495	0.565

Nickel (M)

0.040	0.044
0.040	0.043
0.041	0.043
0.042	0.043
0.042	0.042
0.043	0.042
0.044	0.041

Obsidian (M)

0.00	1.00
0.00	0.00
0.00	0.00
0.00	0.00
0.00	0.00
0.00	0.00
0.00	0.00

Orange (I)

0.00	1.00
0.11	1.00
1.00	0.64
1.00	0.71
0.00	0.79
0.36	0.86
0.00	0.93

Purple (I)

0.01	0.10
0.01	0.40
0.01	0.58
0.01	0.65
0.01	0.68
0.01	0.69
0.01	0.70

Red Felt (E)

0.283	0.646
0.214	0.727
0.498	0.742
0.668	0.750
0.575	0.755
0.456	0.759
0.491	0.763

Ruby (M)

0.05	0.18
0.05	0.23
0.05	0.24
0.05	0.26
0.05	0.27
0.06	0.27
0.10	0.25

Rust (E)

0.750	0.914
0.854	0.921
0.866	0.927
0.878	0.931
0.890	0.936
0.898	0.940
0.906	0.945

Silver (M)

0.00	0.00
0.00	0.00
1.00	0.00
1.00	0.71
0.80	0.79
0.80	0.86
1.00	0.93

UNC Blue (I)

0.312	0.427
0.355	0.433
0.370	0.438
0.385	0.442
0.400	0.445
0.409	0.445
0.418	0.443

Stainless Steel (M)

0.073	0.121
0.073	0.106
0.073	0.108
0.073	0.111
0.073	0.101
0.115	0.135
0.111	0.307

Olive Drab (M)

0.25	0.50
0.17	0.56
0.13	0.60
0.10	0.60
0.07	0.60
0.09	0.60
0.24	0.60

Orange Gladiolus (E)

0.874	0.823
0.879	0.812
0.872	0.800
0.864	0.787
0.857	0.767
0.847	0.750
0.835	0.733

Teflon (M)

0.463	0.718
0.543	0.723
0.586	0.728
0.629	0.733
0.671	0.738
0.708	0.743
0.713	0.748

Tin (M)

0.410	0.491
0.410	0.510
0.410	0.525
0.410	0.540
0.410	0.555
0.437	0.570
0.464	0.592

Tungsten (M)

0.851	0.913
0.855	0.917
0.869	0.922
0.882	0.923
0.896	0.923
0.903	0.924
0.908	0.924

Vinyl (M)

1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00

White (I)

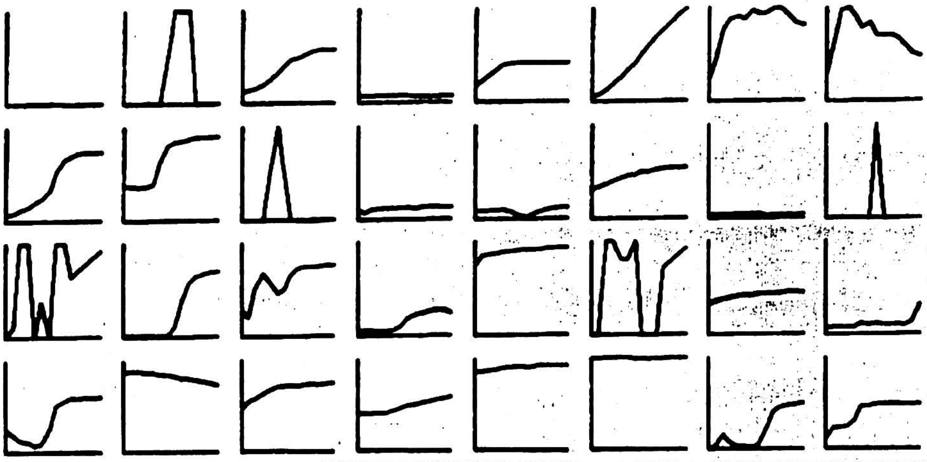
0.01	0.05
0.03	0.20
0.15	0.40
0.07	0.45
0.03	0.48
0.03	0.49
0.03	0.50

Wine Gladiolus (E)

0.10	0.49
0.23	0.49
0.23	0.49
0.24	0.49
0.30	0.49
0.46	0.49
0.48	0.49

Yellow Gladiolus (E)

These thumbnail graphs plot amplitude vertically against wavelength horizontally. The order of the plots left-to-right, top-down, is the same order in which they are presented left-to-right, top-down, on Pages 9, 10, 11, except that the first chart (giving the sampling wavelengths) has been ignored. Thus, the upper-left plot is Blackbody, to its right is Brick Red, the first plot in the second row is Copper, and Yellow Gladiolus is in the lower-right corner.



Call for Contributions

There are many ways that your participation can make this newsletter an exciting and useful enterprise. Any well-written contributions of relevance to the community are appropriate. Here are some possible categories that would make good contributions. You needn't write a whole article; a brief note would be fine; for example a list of useful but obscure books, or a list of the features of your ray tracer, as below.

Program Descriptions

What are the capabilities of your ray tracer? Interesting capabilities include what objects are available for intersection, anti-aliasing techniques, and efficiency methods. How do you describe your databases? Do you support motion? If you use transformation trees, do you provide for topological restructuring of the tree? How do you handle color? What parameters do you use to describe surface physics, and how do you specify the shading for each surface? Do you support textures? If you handle polygons, do you interpolate normals before shading? Do you support constructive solid geometry? Can objects change over time? What kinds of light sources do you support?

Light Extinction Coefficients

Even simple light extinction models seem to work well; for example $I = td$ for constant density d over a path of length t. What are good values of d for different materials? A list of these extinction coefficients for different materials (air, glass, smoky room) would help us model these effects without tedious trial and error. One experimenter's results (coefficients and pictures) would be a useful reference table.

Textures (procedural and table) and Surface Spectra

Not everyone has access to sophisticated scanners. Are there any folks willing to make available one or more high-quality scanned-in textures? Good candidates might include wood, snakeskin, and stone. I'd be willing handle electronic mail distribution of texture files with whatever seems the most practical mechanism.

Has anyone come up with some really nice little procedural textures they'd be willing to share? How about actual code fragments, such as solid textures for wood or marbled rock, or two-dimensional textures for rough brick or stucco?

If you have some nice surface spectra (with any amount of accuracy) sharing them would help enhance our libraries and images.

Survey Articles

Texture mapping is an interesting subject. A survey of 2-d parameterizations of 3-d surfaces would put all the equations and diagrams in one place. Even special cases (such as spheres) can be interesting; compare Mercator, Hyperbolic, Dymaxion, and other projections - why does the United States Geological Survey use Polyconic projection for its topographical quad maps?

Position Papers

A defended position paper can be stimulating reading. Do you have a point of view that isn't in the mainstream, that you believe should be given wider recognition? Write a position paper and argue your case persuasively to your colleagues.

Useful Data

Have you collected useful physical data or experimental values that others might find useful? Tables of useful data (such as the spectra in this issue) would enhance the realism of everyone's images.

Tricks and Hacks

We all have little collections of tricks and hacks. Code tricks can reduce programming time, or maybe speed up run time. Some modelling tricks give cheap pseudo-CSG, or fillets. Share one or two of your favorite hacks in a short note for this new column.

Something Else

Do you use an interesting technique that is published but not well known? A summary paper would help bring it to everyone's attention. Perhaps you have an idea you'd like to see done, but you don't have the time to follow up on yourself. Here's a good way to spread the word and encourage others to investigate it instead.

Deadline

The submission deadline for issue Number 2 is Friday, 4 December 1987. This is when the material must be in my hands in the Netherlands, either via electronic mail or physical delivery. If you're sending physical mail, don't forget to write the customs declaration on the outside to speed the package's journey. See the fine print on page 1 for details about submitting material.

The Mailing List

Here's the mailing list for the first issue. Electronic mail addresses are verbatim as I received them. If you want to receive further issues but your name is not on this list you'll need to write to me with your name and address; this is true even if you received this issue in the mail. Names listed until the double line were participants at the '87 roundtable in Anaheim.

Andrew Glassner (Editor)
before 20 Dec 87
Delft University of Technology
Faculty of Mathematics & Informatics
Julianalaan 132
2628 BL Delft
The Netherlands
`mcvax!dutrun!frits`
after 20 Dec 87
Department of Computer Science
UNC-Chapel Hill
Chapel Hill, NC 27514
`mcnc!unc!glassner`

Rick Speer
PO Box 2651
Seattle, WA 98111

John Francis
Apollo Computer
270 Bellerica Road
Chelmsford, MA

John Peterson
University of Utah
Department of Computer Science
Salt Lake City, Utah
`PETERSON@CS.UTAH.EDU`

David Kirk
Apollo Computer
270 Bellerica Road
Chelmsford, MA
`DAVE@APOLLO.UUCP`

Jim Arvo
Apollo Computer
270 Bellerica Road
Chelmsford, MA

Alan Norton
IBM Research
PO Box 704
Yorktown Heights, NY 10598

Timothy L. Kay
Caltech 256-80
Pasadena, CA 91125
`tim@csvax.caltech.edu`

Paul S. Strauss
Box 1910
Brown University
Providence, RI

Ben Trumbore
120 Rand Hall
Cornell University
Ithaca, NY 14853
`wbt@squid.tn.cornell.edu`

Jeff Goldsmith
JPL 510-264
4800 Oak Grove Drive
Pasadena, CA 91109
`JEFF%WEGA@HAMLET.CALTECH.EDU`

Linda Roy
JPL 510-264
4800 Oak Grove Drive
Pasadena, CA 91109

Frederick Fisher
DEC ML05-2/B6
146 Main Street
Maynard, MA 01754
`DECWRL::JACOB::FISHER`

Paul Heckbert
Pixar
PO Box 13719
San Rafael, CA 94913
`ucbvax!pixar!ph`

Pat Hanrahan
Pixar
PO Box 13719
San Rafael, CA 94913
`ucbvax!pixar!pat`

Michael R. Kaplan
Dana Computer
550 Del Ray Ave.
Sunnyvale, CA 94086
`hplabs!dana!mrk`

Don Marsh
801 Waverly, #4
Palo Alto, CA 94301
`dmarsh@degas.Berkeley.EDU`

Olin Lathrop
Apollo Computer CHF-02-RD
330 Billerica Road
Chelmsford, MA 01824
`OLIN@APOLLO.UUCP`

Eric Haines
3D-Eye, Inc.
410 East Upland Rd.
Ithaca, NY 14850
`hpfcla!hpfcrs!eye!erich@hplabs.HP.COM`

Leonard McMillan
AT&T Pixel Machines
Holmdel, NJ 07733

Darwyn Peachey
Department of Computational Science
86 Commerce Building
University of Saskatchewan
Saskatoon, Canada S7N 0W0
`ucbvax!sask.BITNET!peacheyd`

Jeff Hultquist
310 Grey Ghost Avenue
San Jose, CA 95111

Doug Turner
Department of Computer Science
UNC-Chapel Hill
Chapel Hill, NC 27514
`decvax!mcnc!unc!turner`

Lee Westover
207 Sitterson Hall
Department of Computer Science
UNC-Chapel Hill
Chapel Hill, NC 27514
`decvax!mcnc!unc!westover`

Roman Kuchkuda
Department of Computer Science
UNC-Chapel Hill
Chapel Hill, NC 27514
`decvax!mcnc!unc!kuchkuda`

Frits Post
Faculty of Mathematics & Informatics
Delft University of Technology
Julianalaan 132
2628 BL Delft
The Netherlands
`mcvax!dutrun!frits`

Wim Bronsvoort
Faculty of Mathematics & Informatics
Delft University of Technology
Julianalaan 132
2628 BL Delft
The Netherlands
`mcvax!dutrun!wim`

Erik Jansen
Department of Industrial Design
Delft University of Technology
Jaffalaan 9
2628 BX Delft
The Netherlands
`dutio!fwj`