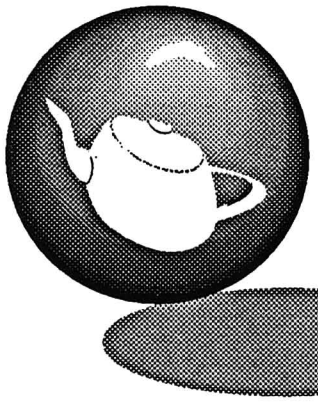


Ray Tracing News

Volume 3, Number 1 "Light Makes Right" May 1989

Table of Contents

 Letters to the editor are welcome, and will be considered for publication unless otherwise requested. Please address all correspondence, including subscription requests, to the editor. Contributions may be submitted on hardcopy, via electronic mail (in plaintext, nroff, or TEX format), or on Macintosh 3.5" disk (in either MacWrite, MS Word, or Text-Only format). Plain text or Text-Only is preferred.

Equations and figures may be submitted in rough form, and will be re-set by the editor if necessary. Editor's address:

Andrew S. Glassner
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
(415) 494 - 4467
glassner.pa@xerox.com

A Ray-Triangle Intersection Algorithm

by Jeff Arenberg

{ucbvax, uscvax} ! trwrb!csed-pyramid!arenberg

In the field of computer generated imagery, the technique of ray tracing can produce very realistic scenes, but at a large computational cost. A ray tracing program spends most of the time calculating the intersection between a mathematical light ray and the mathematical description of the scene's geometry. This article describes an algorithm to efficiently calculate the point of intersection between a ray and a planar triangular surface in $\Re^3$. This algorithm will also find the barycentric coordinates of the intersection point, which is useful for surface normal interpolation and shading calculations.

A triangle in $\Re^3$ must be described by exactly three vectors. In this algorithm, the first vector, $\mathbf{P}_0$, traverses from the coordinate system origin to one of the three vertices and the other two vectors, $\mathbf{P}_1$ and $\mathbf{P}_2$, traverse from the first vertex to the other two vertices. This is illustrated in Figure 1.

A ray P is described by the vector equation $P = O + Dt$.

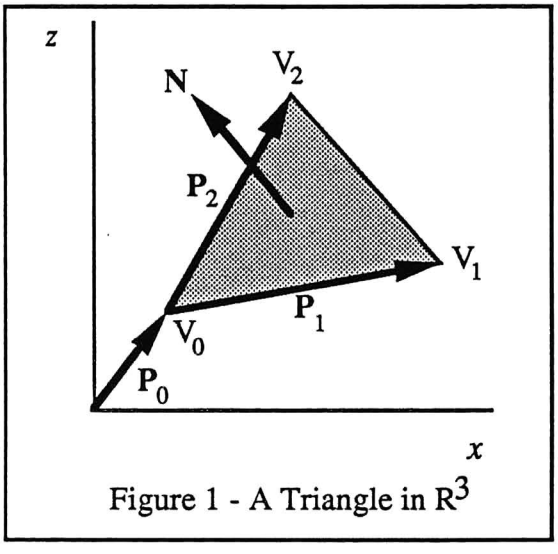


Figure 1 - A Triangle in R^3

For each triangle in the scene, the following preprocessing is performed. The normal vector is found as:

$$\mathbf{N} = \frac{\mathbf{P}_1 \times \mathbf{P}_2}{|\mathbf{P}_1 \times \mathbf{P}_2|}$$

The three vectors, $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{N}$, are linearly independent and therefore constitute a basis of $\Re^3$ [Strang80]. A transformation matrix T is constructed as shown below and the inverse of T is calculated and stored for later use.

$$\text{construct } T = \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{N} \end{bmatrix}$$

$$\text{find } T^{-1} = [\, \mathbf{V}_0^T \,|\, \mathbf{V}_1^T \,|\, \mathbf{V}_2^T \,]$$

and save $\mathbf{V}_i$

At this point the vectors $\mathbf{P}_1$ and $\mathbf{P}_2$ may be discarded, so the storage requirement of this algorithm, $\mathbf{P}_0$ and the three $\mathbf{V}_i$ vectors, is only one vector more than the minimum description of the triangle.

Premultiplying by the matrix, T^{-1}, transforms vectors in the normal $\Re^3$ space to a new space, labeled $\Re'^3$. This vector space has the special property that the vectors $\mathbf{P}_1$ and $\mathbf{P}_2$ transform to the coordinate axes x' and y' in $\Re'^3$ while the normal vector, N, transforms to z'. In $\Re'^3$ space

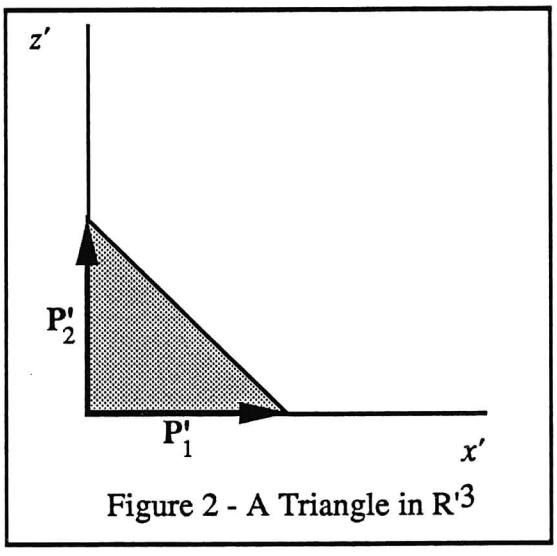


Figure 2 - A Triangle in R'3

the triangle appears as shown in Figure 2. The origin of $\Re'^3$ is shifted from the origin of $\Re^3$ by $\mathbf{P}_0$.

Note, each triangle has a different $\Re'^3$ space associated with it but, in their respective $\Re'^3$, all triangles appear as pictured above. In this space, the intersection calculation is reduced to the straightforward task of finding a point on the x'/y' plane.

The intersection calculation proceeds as follows:

$$\mathbf{O}'_z = \mathbf{V}_2 \bullet (\mathbf{P}_0 - \mathbf{O})$$

if $\mathbf{O}'_z = 0$ return *FALSE*

$$\mathbf{D}'_z = \mathbf{V}_2 \bullet \mathbf{D}$$

if $\mathbf{D}'_z = 0$ return *FALSE*

If $\mathbf{D}'_z = 0$, then the ray is parallel to the plane of the triangle in both $\Re^3$ and $\Re'^3$. If $\mathbf{O}'_z = 0$, then the ray origin lies on the plane of the triangle. In either case, the routine returns, otherwise the value of t is found as:

$$t = \frac{\mathbf{O}'_z}{\mathbf{D}'_z}$$

Because T^{-1} is a linear transformation, the calculation of t is invariant over the change of basis. That is, the ray will intersect the triangle in both $\Re^3$ and $\Re'^3$ for the same value of t. This allows for an additional check to be performed if a global best value of t has been determined, in addition to checking the sign of t.

if $t < 0$ or $t > t_{best}$ return *FALSE*

The intersection point in $\Re^3$ is found from the original definition of the ray :

$$\mathbf{P} = \mathbf{O} + \mathbf{D}t$$

The x' and y' coordinates of the intersection are now found by transforming the point P to $\Re'^3$. The intersection point is inside the triangle if and only if x' > 0, y' > 0 and x' + y' < 1.

$x' = V_0 \bullet (P - P_0)$
if $x' < 0$ or $x' > 1$ return *FALSE*

$y' = V_1 \bullet (P - P_0)$
if $y' < 0$ or $x'+y' > 1$ return *FALSE*

If the algorithm makes it past the last condition, then the ray does indeed intersect the triangle. Shading algorithms that perform either normal vector or intensity interpolation require the barycentric coordinates of the intersection point relative to the vertices of the triangle. These coordinates are found from the $\Re'^3$ intersection point as:

$$BC_0 = 1 - x' - y'$$

$$BC_1 = x'$$

$$BC_2 = y'$$

where the coordinate BC_0 is related to the triangle vertex defined by P_0, BC_1 to P_0+P_1 and BC_2 to P_0+P_2.

If a given ray does intersect the triangle under consideration, then this algorithms will have computed 4 vector sums, 4 vector dot products, 1 vector scaling and 1 scalar division. This does not necessarily conform to fewer floating point operations than any of the other documented algorithms, although all of the operations involved here will benefit from implementation on parallel processing hardware. The real advantage of this algorithm, however, is when the ray does not intersect the triangle, and in most ray traced images, the vast majority of ray/triangle combinations considered fail to intersect. In these cases, it is guaranteed that at least one of the algorithm's numerous test conditions will be true and the routine will exit early, saving the algorithm from unnecessary computation.

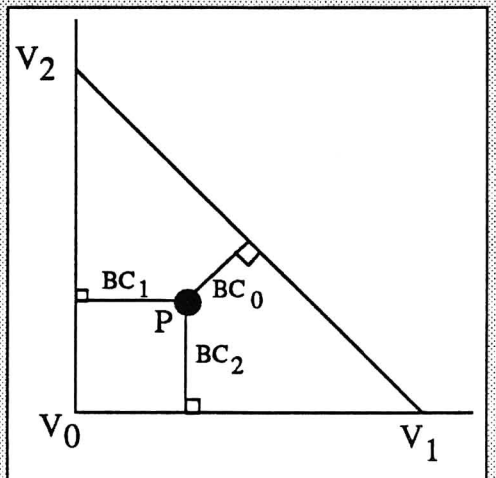
The algorithm described in the article determines if a ray intersects a planar triangle in $\Re^3$. The storage requirement is one vector more than the minimum required to define a triangle, the computational requirement is modest and the algorithm will always take less time if the ray and triangle do not intersect. Thus, this algorithm is well suited to use in the field of image generation by ray tracing. ●

References

[Strang80] "Linear Algebra and Its Applications", Gilbert Strang, Academic Press, New York, 1980.

Editor's note: Two articles in this issue of *The Ray Tracing News* discuss triangles in terms of barycentric coordinates. In a barycentric system, a point is determined by three coordinates.

There are many possible geometric and algebraic interpretations of barycentric coorinates. I have found that one of the most intuitively satisfying is to think of the values as the closest distance of the point to the lines forming each edge of the triangle. To find this distance, erect a perpendicular to the line through each edge, such that the line also passes through the point of interest. In the terminology of the preceding article, the value BC_0 gives the distance to the line formed by vertices V_1 and V_2, and similarly for the other coordinates. The following figure presents these geometric relationships.



$$P = BC_0P_0 + BC_1P_1 + BC_2P_2$$

Geometrical Interpretation of Barycentric Coordinates

Barycentric Co-ordinates for Ray-Triangle Intersections

by Rod G. Bogart
(bogart@cs.utah.edu)

Another way to think of barycentric coordinates is by the relative areas of the subtriangles defined by the intersection point P and the triangle vertices.

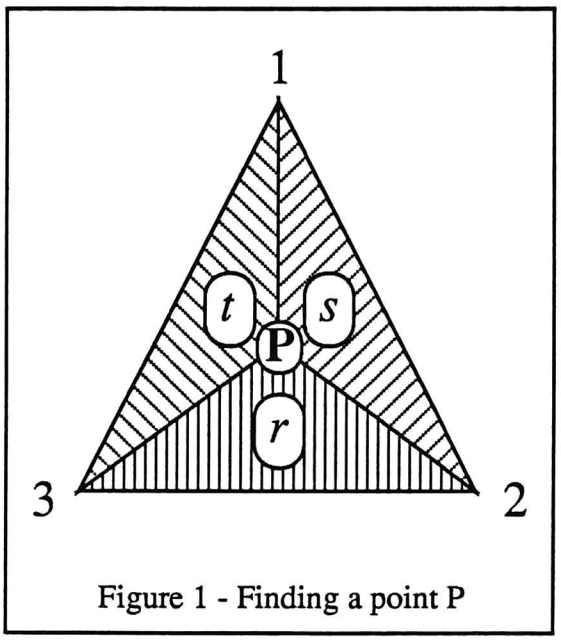


Figure 1 - Finding a point P

If the area of triangle 123 is A, then the area of P23 is *r*A. Area 12P is *s*A and area 1P3 is *t*A.

With this image, it is obvious that *r*+*s*+*t* must equal one. If *r*, *s*, or *t* go outside the range zero to one, P will be outside the triangle.

By using the above are relationships, the following equations define *r*, *s* and *t*.

$$\mathbf{N} = \text{triangle normal} = \overrightarrow{12} \times \overrightarrow{13}$$

$$s = \frac{\left(\overrightarrow{1P} \times \overrightarrow{13}\right) \bullet \mathbf{N}}{|\mathbf{N}|}$$

$$t = \frac{\left(\overrightarrow{12} \times \overrightarrow{1P}\right) \bullet \mathbf{N}}{|\mathbf{N}|}$$

$$r = 1 - (s + t)$$

In actual code, it is better to avoid the divide and the square root. So, you can set *s* equal to the numerator, and then test if *s* is less than zero or greater than $|N|^2$. For added efficiency, preprocess the data and store $|N|^2$ in the triangle data structure. Even for extremely long thin triangles, this method is accurate and numerically stable. •

D-Sampling: An Additional Heuristic for Adaptive Sampling

by Andrew Woo

Dept. Computer Science, University of Toronto, Toronto, Ontario M5S 1A4
andreww@dgp.toronto.edu

One particular anti-aliasing technique in ray tracing is named *adaptive sampling* [Whitted80]. It is basically a heuristic and faster version of supersampling, and the general technique works as follows: for each pixel, point samples are taken at the corners of the pixel. If their colour values differ by some small tolerance (user specified), further point samples are necessary. Then the pixel is divided into four quadrants, and each quadrant needs sampling information at their respective corners. Further sampling within the quadrant is necessary if their corner samples differ by the tolerance. Again, additional four sub-quadrants are necessary.

In summary, this is a recursive approach, creating a hierarchy of quadrants as needed. This sampling is usually halted with the provisal of the following criteria: the maximum recursive depth of the sampling is reached, or the corner sample's colour values differ by less than the tolerance. Combined with jittering [Cook86], this anti-aliasing is usually sufficient.

Motivation for D-Sampling

The main problem with adaptive sampling is that the recursive procedure may be halted due to the second criterion while there still remains a great deal of activity within the quadrant not sampled. Thus the resultant image may not reflect the true database.

A visible example of this problem comes from an infinite plane with an associated checkerboard texture. If the plane is parallel to the X-Y axis and the eye view direction is only some small degree off the X-Y axis, then the checkerboard further off into the infinite horizon will appear pretty lousy.

D-Sampling

From this problem, it is necessary to have another criteria for halting adaptive sampling. We refer to this as *D-sampling*, where *D* stands for distance. The larger the hit distance, the more subdivision into quadrants are done (ie. automatically more samples even when the corner samples differ little). This is done in realization that the probability of having missed activity between corner samples is proportional to the cross-sectional area generated by the four corner samples. And assuming a perspective view, a frustum can be formed from the eye view and the four samples. The cross-sectional area of the frustum increases as it gets farther away.

A good approximation of this cross-sectional area evaluation is the *hit distance*, available from ray-surface intersection calculations. Actually, the maximum distance among the four samples is taken as the hit distance. If all hit distances are large, then by our new criteria, further subdivision and sampling is necessary. If the difference between the corner sample hit distances is large,

then the second criterion should require further subdivision. If not, we can again force further subdivision in the event of drastically changing surface slopes.

We repeat this for reflection and refraction rays. The total sum of the distance of the entire path travelled by the rays is taken as the hit distance. If there are few bounces, then it just boils down to the same reasoning as the above paragraphs. If there are many bounces, then it is likely that a lot of activity lies between samples.

D-Sampling Applied to Filtering

The D-sampling approach can also be applied to filtering. Currently, the filtering in our implementation takes a weighted average of the 8 neighbouring colour values (a 3x3 Bartlett window).

Filtering usually has little effect if the hit distance is sufficiently small. This is assuming that close-by hits are sampled correctly already. Thus filtering can be avoided for such cases.

Filtering also does quite poorly when neighbouring pixels are not properly sampled. It seems to work best when the colour values represent a good approximation to the true colour values. With large hit distances, adequate (but not necessarily sufficient) sampling should be available using D-sampling. Thus with the inclusion of filtering, it should improve the appearance of the image even more.

No Guarantees

This extension is merely a quick hack, and it is not intended to solve all aliasing problems. The hit distance tolerance can be a user specified value. For most images, this heuristic does not do any additional work (negligible), and the image will look no different. However, for some images (like the checkerboard), this should improve the image appearance without having to resort to supersampling. A good example of this improvement is apparent when rendering the infinite checkerboard described above. ●

References

[Cook86] R. Cook, "Stochastic Sampling in Computer Graphics", *ACM Transactions on Graphics*, 5(1), January 1985, pp. 51-72.

[Whitted80] T. Whitted, "An Improved Illumination Model for Shaded Display", *Communications of the ACM*, 23(6), June 1980, pp. 343-349.

Overlapping Boxes

(A sample Graphics Gem in the Efficiency category. See the call for contributions to **Graphics Gems** on page 26.)

Suppose you have two 2d boxes, A and B, and you want to know if they overlap. Assume that the origin is in the lower-left, so that $A_{top} > A_{bottom}$, and $A_{right} > A_{left}$. A and B overlap iff all four of the following relations hold:

$$A_{left} < B_{right}$$

$$B_{left} < A_{right}$$

$$B_{top} < A_{bottom}$$

$$B_{top} < A_{bottom}$$

In 3d, add the following two relations (assuming $A_{far} > A_{near}$):

$$A_{near} < B_{far}$$

$$B_{near} < A_{far}$$

A Review of Multi-Computer Ray Tracing

by David A. J. Jevans

University of Calgary , Department of Computer Science , Calgary, Alberta T2N 1N4
uucp: ...{ubc-cs,utai,alberta}!calgary!jevans

Abstract

Major works in the field of parallel ray tracing are reviewed. A new technique for multi-computer ray tracing is outlined. Future trends are discussed.

Key Words: ray tracing, multi-computer, multi-processor, space subdivision, load balancing, memory balancing, optimistic.

1 Introduction

Scenes are getting larger and more complex, films are getting longer, and the need for high quality rendering has never been greater. Ray tracing provides an elegant but computationally expensive solution. Multi-computer ray tracing provides a means for speeding up ray tracing in a cost effective manner.

Most early research focused on hardware for multi-computer ray tracing. With the recent proliferation of commercial multi-computers current research is directed toward the software problems of parallel ray tracing. Research efforts are primarily directed towards solving problems of load and memory balance.

2 Methods

There are two reasons to consider multi-processing for ray tracing. The first and most obvious is to take advantage of parallel computation to speed up rendering, as suggested by Whitted [Whitted80]. Many small processors can be networked to provide performance equal to or exceeding that of a much larger and more powerful machine.

The second reason is to take advantage of distributed memory. As scene descriptions become more complex this is becoming a more important feature of multi-processor ray tracing. Realistic animation sequences often have scenes consisting of hundreds of thousands or even millions of polygons. To keep these in memory requires an expensive single processor with huge amounts of RAM and disk space. It is often more cost effective to use many smaller less expensive processors with small amounts of memory.

3 Frame Parallelism

Perhaps the most simple method for taking advantage of multi-processors for ray tracing is to render multiple frames on multiple computers. This is very effective when creating animation, although it provides no speedup for rendering a single frame, and provides no access to distributed memory [Leister88].

4 Hardware Solutions

Early research into multi-computer ray tracing focused on hardware solutions.

4.1 LINKS-1

A simple and obvious way of using multi-computers to speed up ray tracing is to duplicate the entire object space on each processor. Nishimura [Nishimura83] used this approach with the LINKS-1. Jobs consisting of regions of the viewing screen pixels are passed to idle processors for rendering. Load balancing is achieved by creating more jobs than there are machines. The LINKS-1 has been used to create numerous ray traced animation sequences.

This method does not take advantage of distributed memory, since the entire object space must be duplicated on each machine. Complex scenes require large amounts of memory on each processor. The method works well, however, on a shared memory multi-processor such as a BBN Butterfly, where a single copy of the scene is available to all processors via the global memory.

Processor utilization on a Butterfly decreases as more processors are added since memory conflicts occur more often. This can be reduced by taking advantage of ray coherence by caching areas of the scene in local processor memory.

4.2 Mesh Machine

Spatial subdivision methods lend themselves to multi-computers since area of space can be distributed among the nodes in the system. Processors are responsible for intersecting rays with the objects that lie inside the areas of space that they have been allocated.

Cleary [Cleary83] suggested partitioning object space with a 3D uniform voxel grid and allocating each voxel to a processor of a mesh connected array. As rays traverse a scene they move through voxels and are passed between processors.

The Mesh Machine, developed at the University of Calgary, was composed of a network of Motorolla 68000 based processors configured as a 2D array. Each processor was connected to its 4 nearest neighbors by a small amount of shared memory, used for passing messages.

4.3 Vector

Plunkett and Bailey [Plunkett85] present an algorithm for vectorizing ray tracing. This doesn't take full advantage of the parallelism inherent in ray tracing and requires costly vector processing hardware.

4.4 Connection Machine

As reported by Crow [Crow88], Karl Sims has written a ray tracing program for the Thinking Machines SIMD Connection Machine. Processing Elements intersect rays with all objects in a scene in parallel. 512 by 512 pixel images consisting of several spheres are rendered in tens of seconds.

4.5 Other

Hardware specific work on algorithms for hypercube architectures and transputer arrays is presented by [Caubet88] and [Bouatouch88]. This

work provides little in the way of new algorithms and may even serve to cloud the important issues in multi-computer ray tracing.

5 Algorithms

Later research has focused on the software aspect of multi-processor ray tracing. These methods utilize distributed memory and processing. Since the object space is not duplicated on all machines, rays must pass from processor to processor as they traverse the scene. This leads to problems of load and memory balancing and reducing the number of messages in the system.

5.1 Dippé and Swensen

Dippé and Swensen [Swensen84] attacked the problem of load balancing by adjusting the subdivision of a scene during the rendering process. A scene is subdivided by a grid of general hexahedra, and these areas are distributed among the available processors. As rays traverse the scene they are passed from processor to processor. Rays must be intersected with the general hexahedra in order to determine which processor to move to; an expensive procedure. If many rays pass through the same area of space, a bottleneck occurs. To alleviate the bottleneck, the shape of the area is changed. This requires altering the shape of neighboring areas, re-sorting objects into the new subdivisions, and possibly passing objects between processors.

If most rays pass through a very small subset of the scene, adjustment of all the areas on all processors may be required. This method balances computational load, but may cause severe memory balancing problems as processor with small areas will use small amounts of memory, and processors near the edge of the grid will contain most of the objects in the scene.

5.2 Nemoto and Omachi

Nemoto and Omachi [Nemoto86] proposed a method for subdividing object space among processors using regular cubes instead of general hexahedra. This method allows for faster traversal of rays through the scene since iterative methods can be used [Fujimoto86] instead of intersecting rays with general hexahedra.

Load balancing is accomplished by sliding the boundaries between abutting cubes. As with Dippé's method, objects which lie inside each adjusted rectangular volume must be re-sorted and passed between adjacent cubes. An advantage to this algorithm is that only one neighboring volume is affected by the sliding operation.

5.3 Kobayashi and Nakamura

A new approach suggested by Kobayashi and Nakamura [Kobayashi87] differentiates between two types of processors in the system: intersection processors, and shading processors. Intersection processors are responsible for intersecting rays with the objects in a sub-volume of space. Rays are passed among intersection processors as they propagate through a scene. When a ray intersects an object, the result is passed to a shading processor. Shading processors generate shade trees, collect illumination information, and output the results.

The object space is subdivided with an octree [Glassner84] and distributed over the intersection processors. If a branch of the octree resides over several processors, vertical traversal requires passing rays between them. In order to reduce vertical traversals, Kobayashi builds an enhanced octree; the adaptive division graph. Octree nodes contain pointers to their brethren and to adjacent nodes. An algorithm for building the adaptive division graph is presented. This method provides memory balancing, but no load balancing algorithms are presented.

5.4 Scherson and Caspary

Scherson and Caspary [Scherson88] adapt hierarchical object extents [Kay86] to multi-computers. The scene's bounding extent tree is duplicated down to a pre-determined level on each processor. Lower levels of the tree are distributed among the processors. When rays begin their traversal of the scene, they may initially be processed through the extent tree by any free processor. If rays can be determined to miss the scene without traversing the tree to a great depth, the load balance on the system remains optimal.

When a ray must traverse the tree to a deeper level than that stored on all processors, it must be passed to the processor which contains the required branch. If many rays must be checked to the full depth of the tree, load balancing can deteriorate. No mechanisms are provided for balancing this computation.

5.5 Pearce

Pearce [Pearce87] implemented the algorithm proposed by Cleary on a network of Corvus workstations and on the Mesh Machine. The object space is subdivided by a regular voxel grid and cubic areas of the voxel grid are allocated to processors. The Cleary [Cleary88] voxel skipping algorithm is used to traverse rays through the grids on local machines. When a ray leaves a voxel grid it is passed to a neighboring processor.

Optimisations to the algorithm are presented, although load balancing was not implemented.

5.6 Kobayashi and Nishimura

Kobayashi and Nishimura [Kobayashi88] present a method which uses regular voxel subdivision, as in [Pearce87], but provides load balancing. The object space is subdivided by a regular voxel grid. Instead of allocating blocks of voxels to processors, voxels are allocated individually among the processors. Adjacent voxels do not necessarily reside on the same processor. Rays are propagated through the scene using the Fujimoto voxel skipping algorithm. As rays pass through voxels they are passed between processors.

Static load balancing is achieved by the allocation of voxels among the processors. If many rays pass through adjacent voxels, the computation does not occur on a single processor, as in the Pearce implementation. A form of dynamic load balancing is achieved by creating "clusters" of processors which all contain the same voxels. When a ray enters a voxel in a cluster, it may be processed by any processor in the cluster.

Message traffic is high in this algorithm since rays must pass from processor to processor as they pass from neighboring voxels. This is wasteful if voxels are empty. Dynamic load balancing occurs only on a cluster level. If all rays pass through the voxels contained on a single cluster, the other clusters may remain idle. The Fujimoto voxel skipping algorithm makes little sense if each voxel resides on a separate processor; each iteration of the DDA algorithm must occur on a different processor. This requires passing the loop variables from processor to processor with each ray. This is not explained in the paper.

An extension of their algorithm [Kobayashi89] is to reallocate voxels among the processors or clusters after each frame of the animation is rendered, according to load balance statistics collected during the rendering.

6 Optimistic Multi-Computer Ray Tracing

A process oriented technique which shares a basis with the Kobayashi and Nishimura algorithm, although developed independently, is presented in [Jevans89]. It uses the principles of optimistic computation and cancellation [Jefferson85]. The object space is subdivided with a regular voxel grid. Vox-

els are allocated randomly among processors. Each processor has a ray generation process which is responsible for rendering some subsection of th pixels. Processors also run many voxel intersection processes which are each responsible for intersecting rays with the objects inside a voxel.

The ray generation processes use the Cleary voxel skipping algorithm to traverse rays through the scene. The Cleary algorithm stores voxels a a one dimensional grid instead of a three dimensional one to reduce storage requirements and allow fast traversal. At each node, instead of a sparse matrix of non-empty voxels, a hashtable is maintained. Entries in the hashtable are machine addresses of intersection processes for non-empty voxels and are indexed by the one dimensional voxel coordinate.

A ray generator starts iterating the Cleary algorithm. The index of the voxel through which the ray is passing is updated each time through the loop. At each iteration, the hashtable is consulted. If the current voxel is non-empty an entry will be found. An intersection request message is sent to the intersection process for the voxel, which performs the intersection test and returns a positive intersection message to the generator if the ray intersected an object.

Generators do not wait for intersection messages to arrive. They optimistically assume that the ray did not intersect with any of the objects in the voxel. The generator continues traversing the ray through the scene, sending intersection request messages to any other non-empty voxels in its path. When the ray has been traversed through th entire voxel grid, a new ray is generated and the process repeats.

If a ray generator process receives a positive intersection message indicating that a ray has intersected an object in a voxel, the optimistic assumtion has failed. All intersection requests made for voxels later in the ray's path should not have been made. These requests must be cancelled.

To enable cancellation, a tree of intersection requests must be maintained for each ray. Whenever an intersection request message is sent, a node is added to the intersection tree. When a positive intersection message arrives, antimessages are sent for all intersection requests that were made for voxels later in the ray's path. Secondary rays for reflection, refraction, and shadowing are fired at this time. These create branches in the intersection tree, and must be cancelled if an earlier intersection message arrives.

Intersection request messages are numbered according to the distance of the voxel along the ray's path. Intersection processors order their message input queues based on this numbering This reduces erroneous computation by encouraging processors to intersect rays with voxels that lie earlier in a ray's path, first. When an antimessage arrives, the corresponding positiv message i deleted from the input queue. If an antimessage arrives after the computation has been performed, it is discarded.

7 Load Balancing

This approach lends itself to several simple and very effective load balancing schemes.

- Static Load Balancing. Since each voxel is associated with its own intersection process, voxels ca be allocated evenly among processors. Randomly allocating voxel intersection processes to processors is a simple means of helping to ensure even load balance among processors. This technique has proven effective fo multi-processor simulations [Nicol88].
- Balancing Ray Generation. As there are many ray generators in the system, a simple method of ensuring an even distribution of load on all processors is to have each generator working on a different area of the screen Since rays fired from pixels on opposite sides of the

screen rarely travel through the same voxels (the opposite side of the ray coherence coin) [Kaplan85] this can help keep all processors busy.

- Process Migration. If a processor becomes too highly loaded in relation to its neighbors, determined by examining the sum of the lengths of its and its neighbor's process input queues, it can migrate intersection processes. When a voxel intersection process is migrated, its objects and input queue must be moved with it to the new processor. Messages which arrive for the migrated proces are forwarded to the new processor and a message is returned to the sending processor informing it of the migration so that further messages can be sent directly.
- Process Cloning. If an intersection process becomes too highly loaded, determined by examining the lengths of the input queues of all processes on a processor, it can clone itself and migrate the clone to another processor. Once this is done it is necessary to inform some processors of the duplicate voxel intersection process in order to split the load. This is accomplished by the original intersection process forwarding some of the ray messages bound for it to the clone. As it does this, it sends a message to the originating processor as to the location of the clone to where its messages have been forwarded.
- Clone Reaping When a clone process determines that it is underloaded it can request to be killed. A message is sent to its parent requesting that the clone be allowed to die and reroute all its messages to the parent. If the parent deems this appr priate, by weighing the request against any other death requests, the process is sent a message allowing it to die. The clone sends its input queue to the parent and dies. The processor which hosted the clone must forward all messages for the clone to the parent, and inform all senders of the routing change. Underloaded parents may request that clones kill themselves and return their input queues to the parent. A process may not die if it has clones of its own which are still alive.

8 Discussion

This method uses algorithms originally developed for distributed simulation; optimistic computation and cancellation. These methods have shown speedup on a variety of parallel processors [Lomow88] [Fujimoto89]. Since ray intersections for a single ray can be processed in parallel, the algorithm introduces a new level of parallelism not previously exploited in multi-computer ray tracers. This, combined with the load balancing techniques described above, allows large processor configurations to be used efficiently.

A prototype of the algorithm is being developed for networks of SUN workstations, the BBN Butterfly, and a Meiko transputer array.

9 Future Trends

Early research into multi-computer ray tracing focused on hardware solutions. Recent algorithmic research has been hardwar independent. This is a trend that should be encouraged. Multi-computers and multi-processors are being produced by numerous manufacturers and are being supplied with operating systems which are advancing to the point that the underlying hardware is no longer visible to the user.

Hardware evolution is clearly directed towards small shared memory multi-processors and large message passing multi-computers. Shared memory multi-processors are typically more expensive than similarly sized multi-computers, and are not available in large configurations, the largest being the 128 node BBN Butterfly [Jenkins89]. Multi-computers are more easily expandable and are

available in much larger configuration such as the 1024 node NCUBE hypercube and 400 node Meiko transputer array.

Load balancing and the often overlooked problem of memory balancing are the key issues in parallel processor ray tracing. Algorithms to tackle these problems must be designed with modern large scale multi-computers in mind. Algorithms must be scalable and, as the numbe of processors available continues to grow, must exploit parallelism not yet apparent in the ray tracing method.

Most of the algorithms reviewed in this article have been simulated and have not been implemented on a multi-computer. The validity of these algorithms remains to be seen. •

10 References

[Bouatouch88] K. Bouatouch and T. Priol, "Parallel Space Tracing: An Experience on an iPSC Hypercube", *New Trends in Computer Graphics*, CG International, 1988, pp. 170-188

[Caubet88] R. Caubet, Y. Duthen, and V. Gaildrat, "VOXAR: A Tridimensional Architecture for Fast Realistic Image Synthesis", *New Trends in Computer Graphics*, CG International, 1988, pp. 135-149

[Cleary83] John Cleary, Brian Wyvill, Graham Birtwistle, and Reddy Vatti, "Design and Analysis of a Parallel Ray Tracing Computer", *Proc CIPS Graphics Interface '83*, 33-34, Edmonton, Alberta, May, 1983

[Cleary88] John Cleary and Geoff Wyvill, "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision", *Visual Computer*, July, 65-83, 1988,

[Crow88] Franklin C. Crow, "3D Image Synthesi on the Connection Machine", *Parallel Processing for Computer Vision and Display*, January, 1988

[Fujimoto86] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications*, 1986

[Fujimoto89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", University of Utah, Dept. of Computer Science, December, 1988

[Glassner84] Andrew S. Glassner, "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, 15-22, October, 1984

[Jefferson85] David Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, 7(3), July, 404-425, 1985

[Jenkins89] Richard A. Jenkins, "New Approaches In Parallel Computing", *Computers in Physics*, January/February, 24-32, 1989

[Jevans89] David A. J. Jevans, "Optimistic Multiprocessor Ray Tracing", *Proceedings of Computer Graphics International '89, 198,*, (In Press)

[Kaplan85] Michael R. Kaplan, "*The Uses of Spatial Coherence in Ray Tracing*", SIGGRAPH '85 Course Notes 11, 1985

[Kay86] Timothy L. Kay and James T. Kajiya, "Ray Tracing Complex Scenes", *Computer Graphics*, ACM SIGGRAPH, 20, 269-278, 1986

[Kobayashi87] H. Kobayashi and T. Nakamura and Y. Shigei, "Parallel Processing of an Object Synthesis Using Ray Tracing", *Visual Computer*, 13-22, 3(1), 1987,

[Kobayashi88] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei, "Load Balancing Strategies for a Parallel Ray-tracing System Based on Constant Subdivision", *Visual Computer*, October, 197-209, 4(4) 1988,

[Kobayashi89] Kobayashi and Nishimura, "Parallel Architecture for Fast Image Synthesis Under Dynamic Environments", *Proceedings of Computer Graphics International '89*, 1989, (In Press)

[Leister88] W. Leister, T. Maus, H. Muller, B. Neidecker, and A. Stosser, "`Occursus Cum Novo', Computer Animation by Ray Tracing in a Network", *New Trends in Computer Graphics*, CG International, 1988, pp. 83-92

[Lomow88] G. Lomow, J. Cleary, B. Unger, and D West, "A Performance Study of Time Warp", *Distributed Simulation*, Society for Computer Simulation, 1988, pp. 50-55

[Nemot 86] K. Nemoto and T. Omachi, "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing", *Proc CIPS Graphics Interface '86*, 43-48, 1986

[Nicol88] D. Nicol, "Mapping a Battlefield Simulation Onto Message-Passing Parallel Architectures", *Distributed Simulation*, Society for Computer Simulation, 1988, pp. 141-146

[Nishimura83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation", *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983

[Pearce87] Andrew Pearce, "An Implementation of Ray Tracing Using Multiprocessor and Spatial Subdivision", University of Calgary, Dept. of Computer Science, 1987

[Plunkett85] D. Plunkett and M. Bailey, "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed", *IEEE Computer Graphics & Applications*, 5(8), August, 1985, pp. 52-60

[Scherson88] I. D. Scherson and E. Caspary, "Multiprocessing for Ray-tracing: A Hierarchical Self-balancing Approach", *Visual Computer*, October, 188-196, 4(4), 1988,

[Swensen84] Mark Dippé and John Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", Computer Graphics, pp. 149-158, 18(3), July, 1984

[Vatti84] Reddy Vatti, "Multiprocessor Ray Tracing", University of Calgary, Dept. of Computer Science, 1983

[Whitted80] Turner Whitted, "An Improved Illumination Model for Shaded Display", *Comm. ACM*, 23(6), June, 1980, 343-349

About this issue of *The Ray Tracing News*...

This issue represents a major redesign of the newlestter. Among the many large and small improvements is a wider inside margin for those who like to punch holes in the pages and save the issues in binders. Thanks to Debra Adams of Xerox PARC for expert typographical advice during the redesign. Thanks also to Dan Murphy of Xerox PARC for the wonderful new masthead.

Articles for this issue of *The Ray Tracing News* were all received via electronic mail, over Usenet, Arpanet, and Bitnet. Articles arrived as plain text, TEX, LATEX, and nroff source. All were converted into Microsoft Word 3.02 where they were edited and basic typography applied.

Layout was performed on a Macintosh II using an E-Systems large-screen monitor. The layout program was Ready, Set, Go! version 4.5 (thus continuing the tradition of using different layout software for every issue!). The main body is 10 point Times on 14 point leading. Headlines are 24 point Avant Garde, bylines are 14 and 12 point Avant Garde, and heads are 12 point Avant Garde. The footers and header are 12 point Helvetica bold.

Illustrations and equations were prepared by the editor. Diagrams were created using MacDraw II and Adobe Illustrator '88 1.8.3. Equations were created with Expressionist 2.0.3, using Bookman, Symbol, and Helvetica faces.

Design, Layout, and Editing by Andrew S. Glassner.

A Proposal for a Hybrid Voxel Traversal Approach

by Andrew Woo

Department of Computer Science,University of Toronto, Toronto, Ontario M5S 1A4
andreww@dgp.toronto.edu

Introduction

Voxel Traversal has proven to be an effective intersection culler for ray tracing [Whit80]. There are basically two variations of voxel traversal: one utilizing uniform-sized voxels (USV) [Fuji86] [Snyd87] [Aman87] etc., and one utilizing variable-sized voxels (VSV) [Glas84] [Kapl85] etc. USV approaches use linear data structures such as the 3d grid, while VSV approaches use hierarchical data structures such as the octree, BSP tree or k-d tree.

The following article intends to look into the pros and cons of these two approaches of voxel traversal. From this, a hybrid voxel traversal algorithm will be proposed so that various advantages from both variations can be retained at little additional cost.

Interpretation of USV and VSV

I perceive VSV as a noble idea, but one that does poorly on many scenes due to the expensive traversal cost, slow access to voxel information, and the preimposed (preprocessed) hierarchy: sometimes the subdivision is not enough, resulting in many intersection tests, and other times, rays never reach the voxels that are deeply subdivided.

I perceive USV as the culler which allows for rapid determination of a candidate set of objects to intersect with (from the voxel's list of object pointers); i.e. a first level culler. In the case of uniformly distributed environments, these candidate sets tend to contain few objects and thusit is sufficient just to intersect against these objects. However, the candidate sets do not always contain a small number of objects due to certain dense regions; which is when USV does poorly.

In the upcoming section, a hybrid voxel traversal algorithm will be proposed. It uses USV as a preprocessed, first level culler and VSV as a lazy evaluated, second level culler. Since it is built on top of USV, it should retain most of the properties and advantages of USV (points 1,2,3). It does not attempt to improve upon the traversal cost (point 5) issue, but attempts to improve on more pressing USV issues, such as dynamic space partitioning (point 4) and handling of sparse environments (point 7).

The Hybrid Voxel Traversal Approach

The hybrid voxel traversal approach uses the USV approach as the ground work. The 3d grid structure is the main data structure to store the uniform voxel information. But within each voxel is an octree containing what we refer to as subvoxels.

Comparison Between USV and VSV

1. USV uses a linear data structure, thus access to information about objects residing in the current voxel is very fast. VSV needs to go through the expensive process of a hierarchy of parent voxels.

2. Traversal cost for USV is very small compared to VSV.

3. Preprocessing for USV tends to be much smaller than VSV.

4. VSV allows for dynamic space partitioning; unlike USV, where the optimal subdivision level is difficult (an analytic solution is presented in [Devi88] but is not accurate for all circumstances) to calculate and is usually user-selected. In addition, the choice for the USV subdivision level is very important since it can severely hamper runtime performance from a poor subdivision choice.

5. VSV can skip over empty space, but it needs to go through a partial hierarch of voxels due to the partitioning method. USV is even more clumzy since it needs to walk through all uniform voxels in the ray's path.

6. VSV should require less storage than USV, especially for sparse environments.

7. USV seems to do very well with quasi-uniformly distributed environments, but does poorly when the environment is overall sparse with localized, dense regions (eg. the football stadium scene as discussed in "Problems with the Octree Subdivision Method for Ray Tracing" by Eric Haines in *The Ray Tracing News* 2(1)); severe subdivision will result in costly traversal, but little subdivision will require a great deal of intersection tests with objects in the dense regions. VSV should do much better under such circumstances.

These subvoxels are constructed only if the voxel contains many objects and if the voxel is traversed many times: i.e. a lazy evaluation approach.

Preprocessing work is done in the identical manner as the USV approach. During the stage of ray tracing, a ray traverses the uniform voxels as before. Upon reaching a voxel containing candidate objects, if the list of objects residing in the voxel is small, then the objects are tested for intersection with the ray and continues on in the usual USV manner. However, if the number of objects in the voxel is large, it is subdivided into eight subvoxel quadrants (an initial octree), and the objects are spatially divided into the eight subvoxels.

Then to find a more refined (and hopefully smaller) candidate set, the octree is traversed in the same manner described by Glassner [Glas84]. This can be done easily with the availability of the t distance at the boundaries of the voxel: the USV implementation using [Aman87]'s approach will guarantee this. If no intersection is found within the subvoxels, then the process proceeds in the usual USV manner onto the next uniform voxel.

With the initial octree within each voxel, the candidate set of objects to intersect with may still be very large. To remedy this, every time the subvoxel visited contains many objects, that subvoxel is further divided into eight quadrants. Thus the octree is subdivided as needed.

Pros and Cons of the Hybrid Approach

Using this hybrid approach, an initial candidate set of objects to intersect with can be quickly determined from the fast traversal through uniform voxels. If the candidate set contains too many objects, then it can be reduced by walking through the subvoxels. Thus scenes containing sparse environments with dense regions should encounter less the trouble. In addition, quasi-uniformly distributed scenes should have about the same runtime performance as USV.

By constructing the octree subdivisions as needed, minimal hierarchy (as opposed to the preimposed hierarchy) suitable for rendering is generated. Dynamic space partitioning is also acquired via lazy evaluation of the octrees. Thus the USV subdivision level problem is not as important, an educated guess for the uniform subdivision level is sufficient. The guess should favou less severe subdivision levels; any insufficiency should be improved by the subvoxels. Thus the need to skip empty voxels is much less critical too.

I also recall that there have been criticisms about the octree not being able to snuggly fit the objects in order to optimize the traversal and intersection processes. However, in our case, th octree is a second level culler and should more than suffice for our purposes.

The main disadvantage of this approach is that more storage is necessary as compared to both USV and VSV. However, the space usage is created only when needed and will not be wasted as in the VSV preprocessing step.

Another Second Level Culler, If You Are Lazy

If the octree second level culler is too much for you in terms of coding, another second level culler can be used. This is exactly described in [Snyd87]: the ray bounding box culler. Basically, the ray span that crosses the voxel can form a bounding box. Intersections are only checked against objects whose bounding box cross the ray bounding box. Each bounding box check requires only 6 flops.

Note, however, some of the ray bounding box checks are wasteful. For USV, the maximum ray span T through any voxel can be calculated:

$$T = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

where Δi are the dimensions of each voxel. A ray span t crossing a voxe only does the ray bounding bo culling if $t<\beta T$, for some suitable $\beta<1$.

This second level culler does surprisingly well for sparse environments under USV. But it only does linear culling and should not be nearly as effective as the above hybrid approach. This is just suggested for the lazy coder.

Final Comments

I have not implemented this approach. It sounds good in theory, but then I don't believe in theory. I would like to hear from fellow Ray Tracing News colleagues as what they think about this approach: improvements, criticisms, etc. ●

References

[Aman87] J. Amanatides, A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", *EuroGraphics '87*, August 1987, pp. 1-10.

[Devi88] O. Devillers, "The Macro-regions: an efficient Space Division Structure for Ray Tracing", Rapport de Recherche du Laboratoire d'Informatique de l'ecole Normale Superieure, Paris, November 1988.

[Fuji86] A. Fujimoto, T. Tanaka, K. Iwata, "ARTS: Accelerated Ray Tracing System", *IEEE Com-*

puter Graphics & Applications, **6**(4), April 1986, pp. 16-26.

[Glas84] A. Glassner, "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics & Applications*, 4(10), October 1984, pp. 15-22.

[Kapl85] M. Kaplan, "Space-Tracing: A Constant Time Ray Tracer", Course Notes on State of the Art in Image Synthesis, SIGGRAPH 85, July 1985, pp. 149-158.

[Snyd87] J. Snyder, A. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", *Siggraph 87*, **21**(4), July 1987, pp. 119-128.

[Whit80] T. Whitted, "An Improved Illumination Model for Shaded Display", *Communications of the ACM*, **23**(6), June 1980, pp. 343-349.

Simple Numerical Root-Finding

Andrew S. Glassner

Xerox PARC, 3333 Coyote Hill Rd, Palo Alto, CA 94304
glassner.pa@xerox.com

(A sample Graphics Gem in the Algorithms category. See the call for contributions to **Graphics Gems** on page 26.)

There are lots of techniques for numerical root-finding, some simple and some very complicated [Press88]. Many graphics techniques that need to find the roots of equations have very modest needs: the equations they need to solve are of one parameter, easily evaluated, and relatively well-behaved, both in their analytic behaviour and numerical stability. Of course, not all algorithms enjoy these characteristics, but there are many that do. For example, consider ray-tracing of low-order polynomical surfaces, finding solutions to cubic curves for spline drawing and selection, and generalized clipping against arbitrary clipping surfaces.

We present here a stable, robust little technique based on [Blinn82] for general root-finding in relatively simple situations, such as those above.

Perhaps the simplest numerical root-finding techniques are Newton-Raphson and Regula Falsi [Pizer83]. Newton-Raphson iterates a single guess of the root; it converges very quickly when it's zeroing in, but it can completely miss a root, and even worse, get stuck in an infinite loop, oscillating around the root. On the other hand, Regula Falsi refines an interval that encloses the root; it's not as fast as Newton-Raphson, but if you start with an interval that includes a root, it's guaranteed to find that root.

A nice combination was suggested by Blinn [Blinn82]. He starts with an interval containing the root, and performs a Newton-Raphson iteration on one end. If the result of that iteration is a new value that is outside of the interval, then he ignores that result and calculates a step of Regula Falsi instead. The result in either case is a new value within the interval, which is used as the new left or right side. The process repeats until the root is trapped to within some precision. The technique is also discussed in [Duff84].

This technique has the drawback that you can waste time computing Newton-Raphson iterations that you then throw away. The hopeis that at some point Newton-Raphson will be close enough that it begins converging; then you save big because of its faster convergance. This whole technique hinges on the ability to quickly evaluate the value of a function (and its derivative, for Newton-Raphson). If function evaluation is expensive in

some application, then this may not be a good approach.

Note also that it is very important that your original left and right values actually trap a root. If they trap several roots the technique will usually find the root with the smallest value, though not always. If you don't trap a root, or your right value is smaller than your left value, the algorithm won't work right. There are no checks for either of these conditions. ●

References

[Pizer83] Pizer, S.M., with Wallace, V.L., "To Compute Numerically", Little, Brown, 1983

[Blin82] Blinn, J.F., "A Generalization of Algebraic Surface Drawing", *ACM TOG*, 1(3), July 1982

[Press88] Press, W.H., et. al., "Numerical Recipes in C", Cambridge 1988

[Duff84] Duff, T., "Numerical Methods for Computer Graphics", Siggraph '84 Course Notes

Here is an implmentation of the root-finding algorithm in Mesa (you can think of this as a high-level pseudo-code, if you like). Notice that the functions **Fofx** and **DofFofx**, which evaluate the function and its derivative, are declared as functions of type FuncProc. Thus they both accept an argument x, and return a real result y; x and y need not be re-declared for each function definition.

In the example functions below, we're simply evaluating x^2-3, which obviously has roots at $\pm\sqrt{3}$.

```
-- FindRoot: roots of univariate functions. Input: f(x), df(x)/dx, and an interval.
FuncProc:       TYPE ~ PROC [x: REAL] RETURNS [y: REAL];  -- Function type declaration

Fofx: FuncProc ~ { y ← (x*x) - 3.0; };   -- F(x)
DofFofx: FuncProc ~ { y ← 2.0 * x; };    -- dF(x)/dx

-- example use: root ← FindRoot[leftx, rightx, tolerance, Fofx, DofFofx];
FindRoot: PROC [left, right, tolerance: REAL, f, fp: FuncProc] RETURNS [root: REAL] ~ {
  newx: REAL ← left;
  WHILE ABS[f[newx]] > tolerance DO
    newx ← NewtonRaphson[f, fp, newx];
    IF newx < left OR newx > right THEN newx ← RegulaFalsi[f, left, right];
    IF f[newx]*f[left] <= 0 THEN right ← newx ELSE left ← newx;
    ENDLOOP;
  RETURN [newx];
  };

NewtonRaphson: PROC [f, fp: FuncProc, x: REAL] RETURNS [REAL] ~ {
  d: REAL ← fp[x];
  IF d # 0.0 THEN RETURN [x - (f[x] / d)] ELSE RETURN  [x-1.0];
  };

RegulaFalsi: PROC [f: FuncProc, lx, rx: REAL] RETURNS [REAL] ~ {
  d: REAL ← f[rx]-f[lx];
  IF d # 0.0 THEN RETURN [rx - f[rx] * (rx-lx) / d] ELSE RETURN [(lx+rx) / 2.0];
  };
```

Here is an implementation of the algorithm in C. The skeleton has been slightly enhanced to directly support polynomials. A global polynomial is used so that NewtonRaphson() and RegulaFalsi() don't need to change to support different kinds of functions; the functions themselves will go the globals for their specific data

It is usually hard to make code simultaneously clear, fast, and concise. I have opted here for clarity. The following code actually runs; I simply copied the working code into the page layout program and formatted it.

One straightforward optimization would be to combine the routines polyroots, NewtonRaphson, and Regula-Falsi into one routine. That routine could also cache the current value of the function at the left end, so it need not be re-evaluated each time, one would need to update that value when the new guess moves the interval's left side.

```
/* simple root-finding, based on Blinn82. Andrew Glassner, 8 May 1989 */
#include <stdio.h>
char *malloc();

double poly(), dpoly();

typedef struct Polynomial_struct {
   int length;              /* the number of coefficients */
   double *coefficients;    /* most significant first, constant last */
   } Polynomial;

Polynomial *CurrentPolynomial = NULL;

double dabs(x) double x; { if (x<0.0) return (-x); return (x); }

double NewtonRaphson(f, df, x)   /* generic Newton-Raphson step */
double (*f)(), (*df)(), x;
{
double d = (*df)(x);
   if (d != 0.0) return (x-((*f)(x)/d)); return (x-1.0);
}

double RegulaFalsi(f, left, right)   /* generic Regula-Falsi step */
double (*f)(), left, right;
{
double d = (*f)(right) - (*f)(left);
   if (d != 0.0) return (right - (*f)(right)*(right-left)/d);
   return ((left+right)/2.0);
}

double poly(x)  /* evaluate global polynomial at x */
double x;
{
double result = 0.0;
double *c = CurrentPolynomial->coefficients;
int i;
   for (i=0; i<CurrentPolynomial->length; i++) {
      result *= x;
      result += *c++;
   }
   return (result);
}
```

```
double dpoly(x)  /* evaluate derivative of global polynomial at x */
double x;
{
double result = 0.0;
double *c = CurrentPolynomial->coefficients;
int i;
   for (i=1; i<CurrentPolynomial->length; i++) {
      result *= x;
      result += (CurrentPolynomial->length - i) * (*c++);
   }
   return (result);
}

double polyroots(left, right, tolerance, f, df, p)
double left, right, tolerance;
double (*f)(), (*df)();
Polynomial *p;
{
double newx = left;
   CurrentPolynomial = p;   /* save current polynomial into global */
   while (dabs((*f)(newx)) > tolerance) {
      newx = NewtonRaphson(f, df, newx);
      if (newx<left || newx >right) newx = RegulaFalsi(f, left, right);
      if ((*f)(newx) * (*f)(left)<=0.0) right = newx; else left = newx;
   }
   return (newx);
}

main() {  /* an example main function to show how this all works */
Polynomial *p;
double left, right, tolerance;
double root;
   p = (Polynomial *)malloc((unsigned)sizeof(Polynomial));
   p->length = 5;
   p->coefficients = (double *)malloc((unsigned)
         (p->length * sizeof(double)));
   /* coefficients are listed most significant first */
   /* This poly is x^4 - 21.14 x^3  - 216.48 x^2 + 3307.22 x - 7693.0
      It has roots at 25, 7, 3.14, and -14
   */
   *(p->coefficients) = 1.0;
   *(p->coefficients+1) = -21.14;
   *(p->coefficients+2) = -216.48;
   *(p->coefficients+3) = 3307.22;
   *(p->coefficients+4) = -7693.0;
   left = -11.0;    /* left and right interval to trap root at 3.14 */
   right = 6.0;
   tolerance = .01; /* iterate until f(root-estimate) < tolerance */
   root = polyroots(left, right, tolerance, poly, dpoly, p);
   printf("root = %f\n",root);
}

/* end of listing */
```

Andrew S. Glassner

Xerox PARC, 3333 Coyote Hill Rd, Palo Alto, CA 94304
glassner.pa@xerox.com

In *The Ray Tracing News* 2(1) (February 1988), Eric Haines contributed an article called "Top 10 Hit Parade of Computer Graphics Books". This article was so popular that I've been prompted to start this new book-review column, *bookshelf*.

I would like to solicit contributions from anyone who has run across an interesting book that they have found useful in their work, and that might prove useful to others. Feel free to submit just a couple of paragraphs about a single book that you like, or send me a bunch of reviews if you feel prolific. I'll attach the name of the reviewer to the review. The review need not be a critical analysis of the book in the traditional sense. If you would simply like to let other folks know of something new and interesting, that's a worthy contribution and I'd be happy to print it.

To start things off, I'd like to describe a few books that I like when thinking about textures and 3-d modeling. These are neat books for inspiring ideas on design and shape, in both 2-d and 3-d. This is by no means an exhaustive list, but rather a few winners from my collection.

A Topological Picturebook
by George K. Francis
Springer-Verlag 1987

This is a very attractive book, filled with gorgeous illustrations. Most of the pictures are hand-drawn with pen and ink, though a few are computer generated, and some others are photos of colored chalk on blackboard. This book is not a text on topology in the traditional sense; the author is not interested in teaching the reader about topology. Rather, he assumes that you already know a good deal of topology, and your interest is in how to draw pictures that show particular topological spaces and situations.

So this volume is really more of an art book than a math book. If you understand topology, you can read the text and better understand the pictures. If you don't know topology, you won't learn it here. This latter class of reader can simply flip through the pages and admire the stunningly unusual shapes and deformations in the illustrations, using them as inspirational material as with any other art or nature book.

Tilings and Patterns
by Branko Grünbaum and G.C. Shepard
W.H. Freeman and Company, 1987

This mammoth book is already a classic in the field. As a mathematics book it establishes a cohesive and comprehensive tiling theory that provokes as many new ideas as it solves. As a source book for design it is filled with fascinating collections of patterns. Anything you ever wanted to know about periodic and non-periodic tiling is here, presented in a lucid and accessible manner.

Those who think that patterns are just a means for creating pretty shapes should be aware that there is a connection between tiling and Turing machines, which means that many investigations of tiling can be interpreted as an analysis of compuation, and vice-versa. Tiling is still a very active field. The three-dimensional crystallography symmetry groups have been well understood for years, but the new field of quasi-crystals (based on an icosahedral symmetry) is gaining a lot of interest with the recent discovery of classes of real crystals that exhibit this form of non-periodic tiling. The bases for both of these topics, as well as many others, are discussed in this book. I like to periodically dip into this book and read a bit or look at some of the pictures.

The Geometrical Foundation of Natural Structure
(A Source Book of Design)
by Robert Williams
Dover Press 1979

This heavily illustrated book describes the author's thinking on two- and three-dimensional geometries. For me, the strengths of the book are in its treatment of two- and three-dimensional tiling and transformations. The manuscript contains lots of polyhedral nets and recipes for close-packing of spheres and polyhedra, as well as some interesting variations on polyhedra. There is also some discussion of transforming some kinds of polyhedra into others. I like this book both as a reference and as a source for inspiration for regular 3-D structures.

Handbook of Regular Patterns
(An Introduction to Symmetry in Two Dimensions)
by Peter S. Steven
MIT Press 1981

This book discusses the point groups, the seven line groups, and the seventeen 2-D symmetry groups that underly many tilings seen in art and nature. Each chapter begins with a short summary of the properties of one group, and then presents many, many pictorial examples of designs based on that group.

The book is printed in black-and-white, though some gray tones are used to indicate additional colors. The examples are taken from the world's artworks, and many are accompanied by descriptive text. For learning about symmetry groups I'd prefer to read a book on group theory or crystallography, but for examples of designs based on symmetry this is an excellent reference.

The Grammar of Ornament
by Owen Jones
Dover Press 1987

This is one of the undisputed classics in the field of ornamental design. Owen Jones (1809-1874) collected bits of artwork from all over the world, illustrating principles of design executed by master craftsmen. The text discusses the sources and techniques of the different categories of ornament, organized along historical, geographical, and sociological themes.

Jones advocates 37 Propositions in "the ar-

rangement of form and colour, in architecture and the decorative arts", which the illustrations in the book are meant to illustrate and amplify. Literally thousands of examples of graphic art, many in full color, create an abundant sourcebook for design and ornament. Dover publishes two versions of this once-rare book: a beautifully produced, deluxe hardcover reproduction of the original manuscript (13-1/2 by 9 inches), and a paperback edition contaning all 100 color plates (about 12 by 9 inches).

Snow Crystals
by W.A. Bentley and W.J. Humphrey
Dover Press 1962

When talking about fun source books for design I just couldn't bear to leave this one out. There's a few pages of introductory text about snow crystals, but the bulk of the book is some 2400 large, black-and-white pictures of snow crystals. Indeed, no two are alike! I like just flipping through the pages of this book every now and then, to remind myself of the diversity that can be found even in such simplicity.

The Power of Limits
by György Doczi
Shambhala Publications 1981

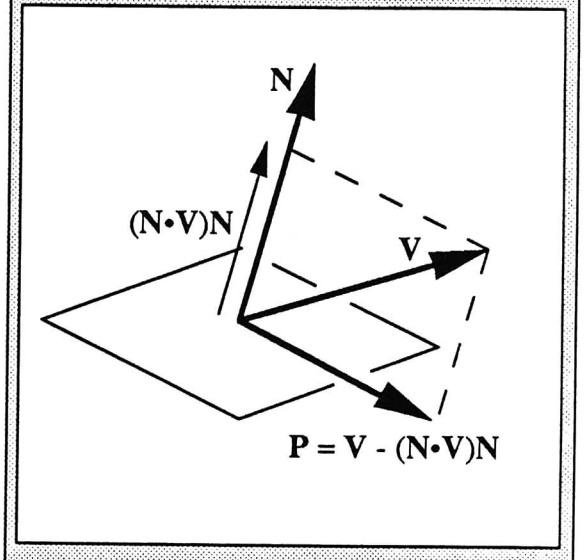
The author of this book is a big fan of harmony, in music, architecture, art, and nature. He found that many of the design elements of aesthetically pleasing structures could be related to each other through geometric ratios. The book is extensively illustrated with structures ranging from human bodies to temples, nautilus shells to dinosaur skeletons, Stonehenge to Chinese pagodas. Many of these illustrations are accompanied by harmonic diagrams that show the relationships of the various pieces of the structure. This is a fun book to come back to every now and then, to remember how prevalent these natural relationships are. ●

Projecting A Vector Onto A Plane

(A sample Graphics Gem in the Geometry category. See the call for contributions to **Graphics Gems** on page 26.)

Sometimes it's necessary or useful to find the projection of a vector onto a plane. As with most geometric problems, you can solve this one algebraically or geometrically.

First, let's look at the geometric approach. Suppose you have a vector **V** which you'd like to project onto a plane with normal **N**; the result is vector **P** (assume all vectors have unit length). From the diagram, we can see that the projection of **V** onto **N** is (N•V)N; this is the component of **V** that is not in the plane. By subtracting this from **V** we find **P = V - (N•V)N**.

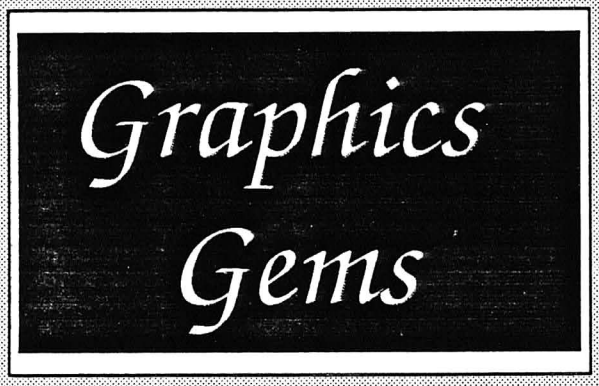


The algebraic approach says that since **P** is in the plane, (1) **N•P = 0**. Since **P** is coplanar with **V** and **N**, (2) $\mathbf{P} = \alpha\mathbf{N} + \beta\mathbf{V}$. Substituting (2) into (1) we get $0 = \alpha + \beta(\mathbf{N}\cdot\mathbf{V})$ (recall **N•N=1**). Setting $\beta=1$, then $\alpha = -(\mathbf{N}\cdot\mathbf{V})$, so we conclude that **P = V - (N•V)N**, as expected. ●

New Book: Call for Contributions

How many times have you re-derived some basic geometric transformation that you're derived a half-dozen times before? Have you ever needed to derive a data structure on the fly, though you're sure that you could make a better one if you had the time?

I find that this sort of thing happens to me again and again. I have to do something that I know has been done before (and probably better) by someone else or myself, but I have to invent a new solution on the spot just to get the project done. I think we need some way for us to share the best little bits and pieces of our discipline with each other.



Announcing a new book to fill that need: **Graphics Gems**. Graphics Gems will be a compilation of the tricks of our trade: the optimized inner loop, the essential snippet of geometry, the algebraic simplification that improves performance. Each Gem will be between 1/2 and 2 pages, and will embody some simple bit of insight that is worth keeping and sharing (explicit source code listings can run longer if needed, but the explanation should still be short). Gems are not research results, and don't necessarily represent deep thinking; they're just a good solution to a graphics problem. Think of how much time and energy you would have saved if you had a book like this when you built your first rendering or modeling package. And imagine how much better that package would have been!

Contributions to Graphics Gems may be very informal. You may submit them in either plaintext, or as input for a common word processor (MS Word, TEX, nroff, etc.), either physically or via email. You may include rough figures; we will redraw them professionally for the text. Appropriate submissions include anything relevant to graphics and graphics programming (note just ray tracing): geometry, algebra, data structures, algorithms, explanations of interesting phenomena, whatever. Please take a few minutes to write down your favorite hacks and insights and send them in. A couple of modest examples to get the ball rolling appear on pages 7, 19, and 25 of this issue.

Contribute - share your knowledge and experience! And please pass along this announcement to your colleagues. Send your submissions to:

Andrew Glassner
Editor, Graphics Gems
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
(glassner.pa@xerox.com)