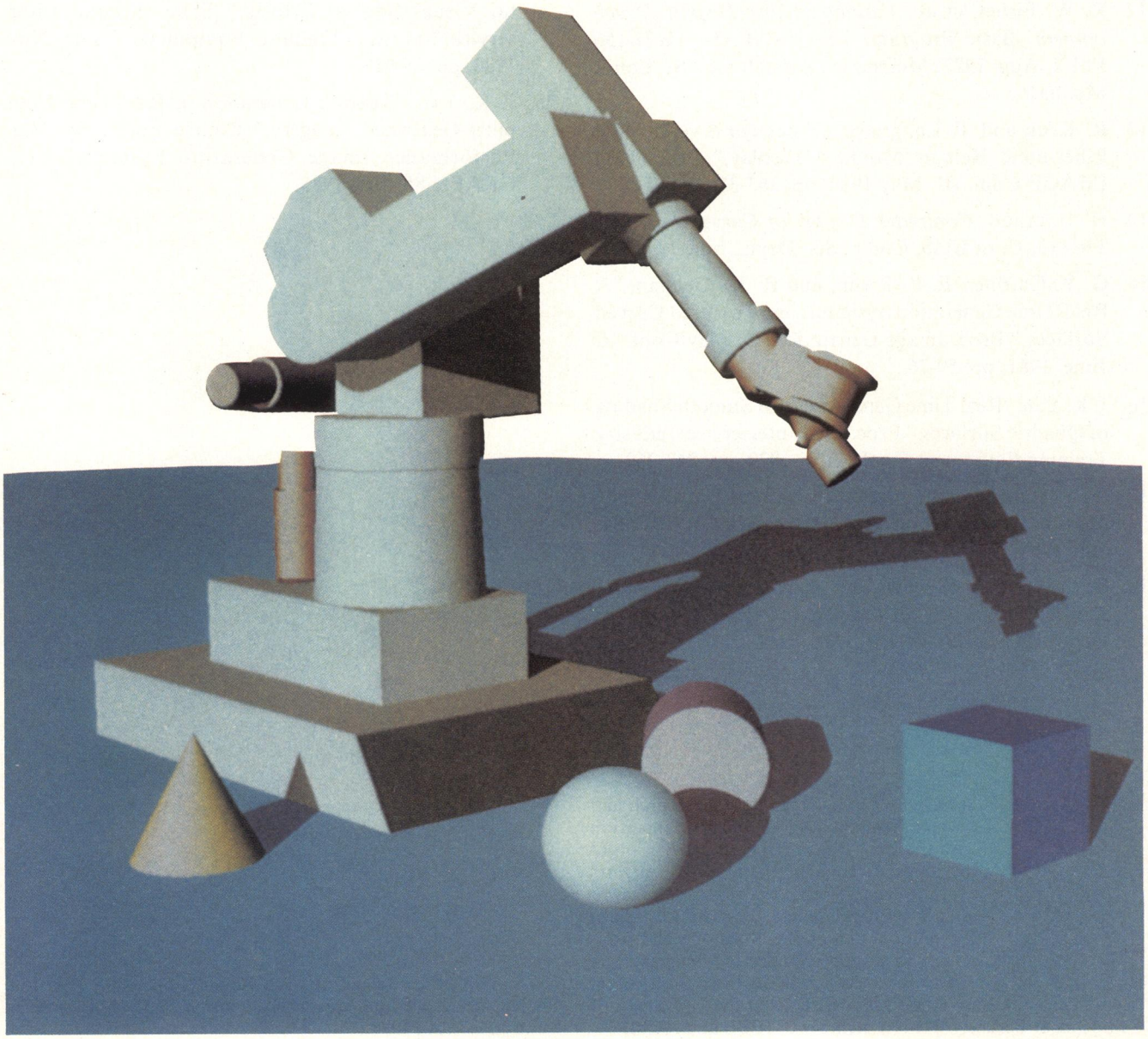
Ray tracing has always been a computational problem. Can it be improved?

The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed

David J. Plunkett and Michael J. Bailey

Purdue University



0272-1716/85/0800/0052$01.00 © 1985 IEEE

Ray tracing is considered the finest technique available for the synthesis of realistic images. No other synthesis method provides anywhere near the same degree of authenticity. Spectacular images, incorporating such special effects as multiple light sources, shadows, reflection, transparency with refraction, and texture mapping have become almost commonplace. And by reducing the image generation problem to its absolutely lowest level of one ray at a time, ray tracing permits the incorporation of a large variety of surface types in a scene.

Recently published work in the area of ray tracing has expanded the state of the art in several ways. More realistic effects have been obtained.[1] More types of surfaces and volumes have been fired upon.[2–4] Ray tracing has been made an integral part of computer-aided design applications.[5–7]

While ray tracing is unsurpassed in the quality of images created, no other technique produces the same exorbitant amount of computing time. Turner Whitted lamented and described these high computation times.[8] The bottleneck was easy to find. He estimated that 75 percent of the time required to synthesize an image was consumed in the process of computing ray-surface intersections. For complex scenes that amount could go as high as 95 percent! Whitted sought to reduce the time by cleverly bounding certain elements of the scene, so that not as many rays would need to be fired. For the same reason, he also implemented an "adaptive pixel subdivision" scheme. Beyond these and other variations on the same efforts, the excessive compute time of the algorithm has generally been grudgingly tolerated as a necessary evil if one wished to enjoy the benefits of ray tracing.

Ray tracing is, without a doubt, the most versatile and dramatic approach to realistic rendering. But its practical use has been severely restricted by the massive computing times required. Research described in the literature indicates that this is largely due to the time required to calculate the intersections of many rays with the surfaces in the scene.The work presented here uses a vectorized computer architecture to speed the ray-tracing process. When a ray needs to be fired, that request is placed on a *ray queue*. When the queue becomes full, the vector processor fires all those rays at once. The program then goes back and uses the intersection results where they were originally requested. Significant speed increases result from this retooling of the algorithm.

This article extends the scope of the ray-tracing technique by directly addressing the computer-time problems. By noting that ray tracing is inherently a parallel process, we will vectorize the ray-tracing algorithm. This is not a matter of simply adding vector syntax to an existing program. The entire process must be rethought. This will allow us to use the full power of the CYBER 205 supercomputer to speed the algorithm dramatically.

Ray tracing for computer-aided design applications

Ray tracing has a history of use in CAD applications. Some of the original development by Goldstein[9] has resulted in MAGI's SynthaVision system. Since then the technique has shown up periodically in several other CAD efforts because of its ability to generate images and mass property information for a variety of geometric forms.[5,6] There is also an assortment of other engineering analyses that take advantage of the technique. Computation of heat-transfer shape factors has been implemented.[7] Applications in radar and antenna analysis have been suggested.

The motivation for the work presented here was the need for rapid production of images and the determination of mass properties for mechanical engineering computer-aided design. The CADLAB's geometric modeling efforts are largely focused on solid modeling of mechanical parts. Constructive Solid Geometry (CSG) is used to generate the geometric database.[10] During the design process, the engineer requires rapid and abundant feedback concerning the effects of the design decisions being made. Current CSG rendering programs are not fast enough to be incorporated in an interactive design process. "Quick-and-dirty" imaging has been achieved by approximating the solid primitives with faceted polygonal surfaces.[11] This gives dramatic speed improvements. However, even large numbers of polygons do not describe a smooth surface exactly. For accurate imaging, the exact surface types must be used. Also, the quick polygonal-approximation schemes do not lend themselves to the variety of special display effects that ray tracing does.

The CDC CYBER 205

Since ray tracing is so computationally expensive, we welcome any help we can get from advances in the state of the art of computer architecture. A recent architecture development that dramatically increases the speed of certain arithmetic operations is the vector, or array, processor. These processors are optimized for the parallel manipulation of arrays (vectors) as opposed to serial manipulation of single data elements (scalars). Exactly the same operations are performed on each data element in rapid succession. The results of a vector operation are not available until the entire vector operation is complete.

The CYBER 205 supercomputer is actually two separate computers connected by a very fast communications link. One of the computers is a vector processor. Only parts of a typical program can execute in the vector hardware. For nonvector operations the 205 also contains a fast scalar machine. Program instructions that execute on the vector half are referred to as vector code. Their complement is scalar, or serial, code.

Scalar CYBER 205 Fortran (FTN200) looks very similar to FORTRAN 77. In addition, the language has extensions that allow the programmer to instruct the machine explicitly to execute certain operations in the vector processor. To the 205 a vector is defined as a contiguous set of memory locations. The language extensions reflect this. For example, the scalar DO loop:

```
      DO 10 I = 1,N
        A(I) = B(I)
   10 CONTINUE
```

can be coded for the vector processor in one statement:

```
A(1;N) = B(1;N)
```

where N is the length of the vector operation. This same method can be applied to arithmetic operations. The loop:

```
      DO 10 I = 1,N
        A(I) = B(I) * C
   10 CONTINUE
```

can be vectorized as

```
A(1;N) = B(1,N) * C
```

Notice that C in this example is a scalar, not a vector. The vector processor recognizes this and treats C as a vector by "broadcasting" its value across each of the operations.

Not all DO loops are candidates for vectorization. In particular, DO loops where a value used in one pass through the loop depends on results of previous passes cannot be vectorized. As an example consider the following DO loop:

```
      DO 10 I = 2,N
        A(I) = B(I) * A(I-1)
   10 CONTINUE
```

The vector processor must have the values of both B(I) and A(I) before it can perform the multiplication. Since the values of A(I-1) are not available ahead of time, the loop cannot be vectorized.

The vector equivalent of the IF-THEN-ELSE statement is another extension of CYBER 205 Fortran. The scalar loop

```
      DO 10 I = 1,N
        IF( A(I) .GT. B(I) ) THEN
          C(I) = A(I) * B(I)
        ELSE
          C(I) = A(I) / B(I)
        ENDIF
   10 CONTINUE
```

can be converted to vector code as

```
      WHERE ( A(1;N) .GT. B(1;N) )
        C(I) = A(1;N) * B(1;N)
      OTHERWISE
        C(I) = A(1;N) / B(1;N)
      ENDWHERE
```

The results of these two sections of code are the same. However, in the scalar version only N arithmetic operations are performed. In the vector example each section of the conditional WHERE statement is computed. The decision about which result to store depends on the result of the vector comparison. Therefore (N + N) arithmetic operations are performed in the vector example. Despite the fact that the vector version requires more arithmetic computations than its scalar counterpart, the actual execution time can be much less.

The execution time depends on two values: a small constant multiplied by the number of elements in the arrays to be processed (N in the above examples) and a start-up time independent of N. Because of the start-up costs, instructions with very small N may actually execute more slowly on the vector processor than on the scalar processor. Although no single break-even point can be quoted, a typical value is in the range of five to ten elements. All vectors longer than this value will execute faster on the vector processor. In general, longer vectors give greater speed increases when compared to the same instructions on the scalar processor. The length of a vector is limited to $2^{16}-1$ by the address conventions used in the CYBER 205.

In addition to vector operators, CYBER 205 Fortran includes vector versions of almost all of the standard Fortran library functions (VSIN, VCOS, VSQRT, etc.). These functions return a vector (array) of value, rather than a single value, and are referenced in a slightly different manner from standard library functions. For example:

```
A(1;N) = VSQRT( A(1;N); N )
```

returns the square root of the first N elements of array A. A syntax extension allows the programmer to write vector functions in FTN200.

Vectorized ray tracing

It should be obvious from the discussion of the previous section that not all algorithms are well suited to the vector processor. There are two fundamental requirements: There must be some opportunity to perform the same calculations on a stream of data elements. Also, each of the individual calculations must be independent of the results of other calculations within the data stream.

The traditional ray-tracing algorithm is a parallel algorithm. The rays that are generated to determine the intensity of one pixel are completely independent of all other rays that have to be traced. Each traced ray determines the surface visible from the origin of that ray. To determine the intensity at one pixel may require that many visible surfaces be found to model the effects of shadows, reflection, and transparency. This algorithm can be vectorized if we perform the same calculations simultaneously in rays that are mutually independent. The rays that result

from one individual pixel are not mutually independent. However, rays belonging to different pixels of the image are completely independent. All rays are potentially being intersected with each surface in the scene. We will now discuss the notion of intersecting an individual surface with many rays in a single vector operation.

It is convenient to think of placing rays in a queue that will not be processed until it reaches some predetermined length. Rays can be placed on the queue from two different sources. One source is the original ray traced through a pixel on the image plane. This ray causes a pixel to become "active." That is, rays are currently being processed that belong to this pixel. If a traced ray hits a visible surface, it can spawn more rays, which must be queued. Thus, the other source of queue rays comes from rays that have just been traced and are now spawning child rays of their own. Rays that are directed at each light source must be traced to determine if the surface is in a shadow. Other rays must be spawned if the surface has reflective or transparent properties.

Using this approach, a vectorized ray-tracing algorithm becomes:

WHILE (there are pixels left to evaluate)
{

1. Until the queue is full, add more rays. Some of these rays originate at the eye position and are directed through new (as yet inactive) pixels.
2. Calculate the intersections of the entire queue of rays with each surface in the scene, one surface at a time, using vector code.
3. Determine which of the above intersections is the visible surface for each ray, using CSG evaluation techniques.
4. From those rays which hit a visible surface, spawn any further rays necessary to model special effects. Add these rays to the queue.
5. Determine the intensity of any pixels that have all their visible surface calculations complete.

}

This algorithm has several constraints. The most obvious is that the program is necessarily more complex than its scalar counterpart, since the properties of more than one pixel are being computed at a time. Information must be kept that indicates which pixels are active and where each ray on the queue belongs (e.g. ray #I on the queue is the ray directed at light source #J fired from visible surface #K). In fact, nothing guarantees that the calculations for the pixels will complete in the order in which they were started. (Consider the number of rays that must be traced for a transparent, reflecting surface compared to an opaque, nonreflecting surface.)

Perhaps a more important constraint is the increased memory requirements of the vector algorithm. The result of the intersection of one ray with all surfaces is a list of the distances to each surface that the ray hit. Because we were working with a CSG geometric-modeling scheme, each surface hit is really part of a primitive volume that has at least two interesting points. Therefore, at least four entries are needed in the hit list for each primitive struck by a ray: two for the distances to each surface and two to identify which surface was hit. In the scalar version of the algorithm the memory requirement for the intersection test is 4N, where N is the number of primitives in the scene. In the vector version, the space required is 4NL, where L is the length of the ray queue. This space is not insignificant! To take maximum advantage of the vector processor we want a long queue to minimize the relative influence of the start-up time associated with using the vector processor. But we do not wish to exceed the size of the real memory of the computer. In this implementation, the queue length, L, was chosen to be 500. In some of our more complex examples, there are approximately 100 primitives. This means that the memory required just to hold the results of one queue intersection is around 200,000 words. Since the Purdue CYBER 205 has one million words of real memory, this was sufficient for us to avoid page faults.

An example: the sphere

As an example of how a vectorized intersection routine could be built, consider the simplest of all CSG primitives, the sphere. Figure 1 shows a sphere centered at $\bar{C}$ with radius R. A single ray is to be intersected with this sphere. The ray's origin is $\bar{P}$. A unit vector in the direction of this ray is $\hat{D}$. We define the vector from the base of the ray to the center of the sphere $\bar{G}$ as

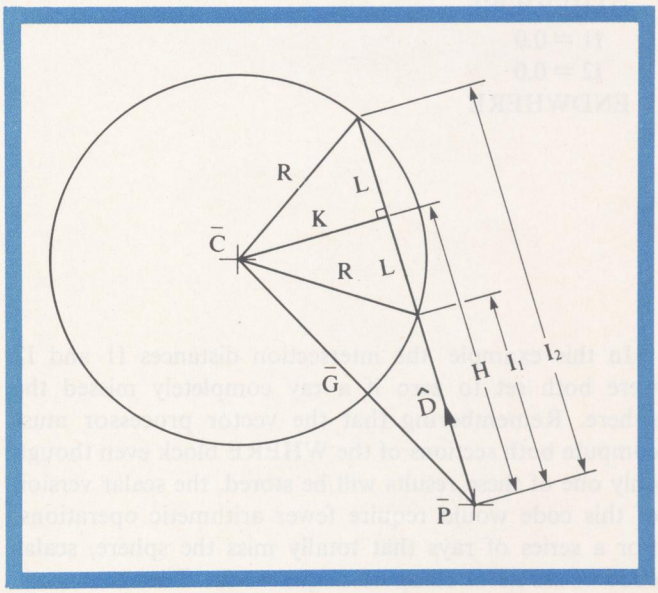


Figure 1. Intersection of a ray and a sphere.

$$\bar{G} = \bar{C} - \bar{P}$$

The distance H is found as

$$H = \bar{G} \cdot \hat{D}$$

K is the third side of the right triangle whose sides are $\bar{G}$, H, and K. The square of K can be compared from the Pythagorean theorem as

$$K^2 = |\bar{G}|^2 - H^2$$

We're now set up to find the distances to the sphere surfaces, I_1 and I_2. The square of distance L is

$$L^2 = R^2 - K^2$$

If L^2 is greater than zero, we know there are intersection points on the sphere. The distances to these two points are then:

$$L = \sqrt{L^2}$$
$$I_1 = H - L$$
$$I_2 = H + L$$

Assuming the ray information ($\bar{P}$ and $\hat{D}$) is stored in arrays (ray #i stored as PX(i),PY(i),PZ(i),DX(i),DY(i), DZ(i)) the vector implementation of this algorithm is

```
GX(1;N) = CX(1;N) - PX(1;N)
GY(1;N) = CY (1;N) - PY(1;N)
GZ(1;N) = CZ(1;N)  - PZ(1;N)
H(1;N) = GX(1;N)*DX(1;N) + GY(1;N)*DY(1;N) +
  (1;N)*DZ(1;N)
GSQD(1;N) = GX(1;N)*GX(1;N) + GY(1;N)*GY(1;N)
  + GZ(1;N)*GZ(1;N))
KSQD(1;N) = GSQD(1;N) - H(1;N)*H(1;N)
LSQD(1;N) = R*R - KSQD(1;N)
WHERE( LSQD(1;N) .GE. 0.0 )
  L(1;N) = VSQRT( LSQD(1;N);N )
  I1 = H(1;N) - L(1;N)
  I2 = H(1;N) + L(1;N)
OTHERWISE
  I1 = 0.0
  I2 = 0.0
ENDWHERE
```

In this example the intersection distances I1 and I2 were both set to zero if a ray completely missed the sphere. Remembering that the vector processor must compute both sections of the WHERE block even though only one of these results will be stored, the scalar version of this code would require fewer arithmetic operations. For a series of rays that totally miss the sphere, scalar code would never compute a square root. The vector code of this example will compute one square root for each of the rays, regardless of whether it is really necessary.* Even so, the vector code is substantially faster than scalar code (see Table 2). But for more complicated intersection algorithms a larger number of extra operations must be performed by the vector processor. This explains the differences in speedup associated with different surface types.

Even though this example is quite simple, the same concepts apply to other surface types. For any intersection, a series of rays is being intersected with one surface. This satisfies the basic criteria for vectorization. We have a series of data items on which we wish to perform the same operations.

Vectorized tree traversal

In the Constructive Solid Geometry process, determining the surface first struck by the ray is not a matter of just intersecting the ray with all primitives in the scene and selecting the closest one to the ray origin. Rather, it is a matter of collecting information on where the ray intersects each primitive and how those primitives have been combined into the overall scene. In CSG modeling such as this, the primitives are combined with the Boolean operations of union, intersection, and subtraction. The semi-infinite ray that was fired penetrates the collection of primitives in a series of line segments. If the same Boolean operations used to make up the scene from the primitives are used to combine that series of line segments, the resulting line segments represent how the fired ray penetrates the actual scene.

Because CSG operations are typically stored in a binary tree data structure, the process of determining how the ray penetrates the scene based on how it penetrates the collection of primitives is called *tree traversal.* In the scalar version of the program, the time to perform the tree traversal was small with respect to the time necessary to determine intersections. But after the intersection code was vectorized, the cost of the tree traversal became larger by comparison (see Figure 3). It then seemed worthwhile to vectorize the tree-traversal step in the overall algorithm too.

The process is not actually interested in all the places that the ray enters and leaves the scene. The process is interested only in where the ray enters the scene the first time. The fired ray has numerous intersections with the primitives in the scene. These are sorted by distance, nearest to farthest. Each intersection must then be examined in turn to determine if it is on the total CSG object.

*We need not concern ourselves with the possibility that at times we will be taking the square root of a negative number. The 205 does not abort a job when such an indefinite number is *produced,* but rather when it is actually *stored.* The WHERE block prevents the number from being stored. The indefinite produced is simply thrown away.

For example, in Figure 2, the first ray intersection (on the square) is not a surface of the CSG object. It's not until the third intersection along the ray, (the second intersection of the square) that the visible surface is encountered.

An intersection point may be evaluated through the CSG binary tree. For example, if two primitives, A and B, have been joined, and a point P is inside of each, then the point P will also be inside of the resulting object A∪B. Table 1 shows the results obtained when objects A and B are joined, intersected, and subtracted; and when the point P is inside, outside, and on the boundary of each.

Table 1. In/Out/On decision table.

Point P vs A	Point P vs B	A union B	A intersect B	A subtract B
In	In	In	In	Out
In	Out	In	Out	In
In	On	In	On	On
Out	In	In	Out	Out
Out	Out	Out	Out	Out
Out	On	On	Out	Out
On	In	In	On	Out
On	Out	On	Out	On
On	On	On	On	On

Each of the many primitive intersection points is evaluated through the CSG tree using the rules in Table 1. The first point where the ray penetrates the CSG object is the first of those sorted points to produce a value of *on* during the CSG evaluations.

The tree traversal can be vectorized if the same operations can be performed simultaneously on the intersections produced by all the rays in the ray queue. This simply implies that we must traverse the tree in the same order for every ray. The same comparisons can then be made at each node as the tree is traversed. The results of each of the comparisons will be different for each ray, but the operations that produce those results will be the same.

Results

The most dramatic results of vectorization appear when comparing scalar-intersection and vector-intersection subroutines alone. Under these conditions the overhead required to store and process the results of the vector code is not present. Table 2 makes this type of comparison, showing the CPU time for intersecting 100,000 rays with different primitive volumes. The vector code intersected 500 rays at a time. The associated speedups (how much faster the algorithm ran) are given in Table 3. Vectorization always netted a speed gain of at least one order of magnitude. The sphere-intersection algorithm was also

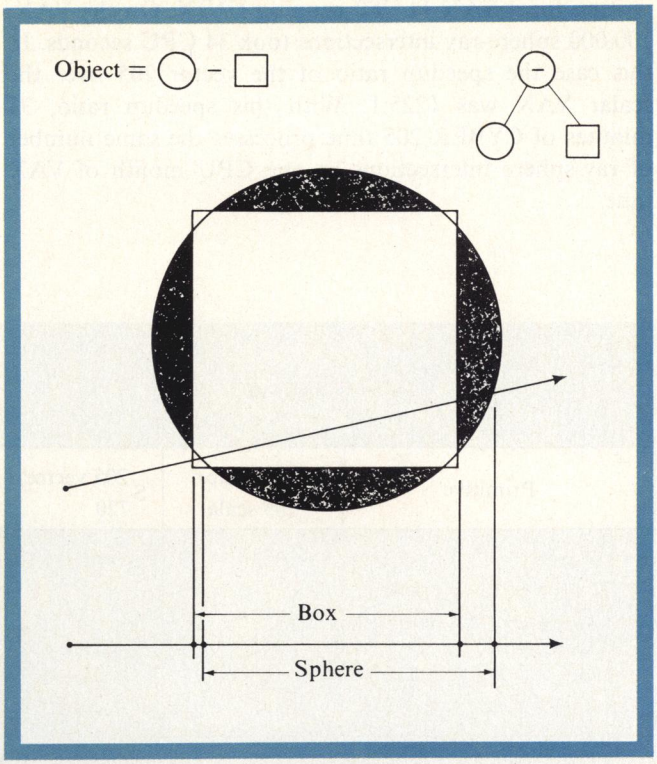


Figure 2. Finding the visible surface of a CSG object.

Table 2. CPU times.*

Primitive	CYBER 720	CYBER 205 Scalar	CYBER 205 Vector
Sphere	13.1	0.94	0.03
Quadric (cylinder, cone, ellipsoid, paraboloid)	51.5	2.73	0.16
Parallelpiped	59.7	3.28	0.20
Torus	137.8	8.40	0.77
Steiner Surface	216.0	11.16	1.05

*CPU times are in seconds, vector length = 500.

written for a VAX 11/780 running UNIX. On the VAX, 100,000 sphere-ray intersections took 34 CPU seconds. In this case the speedup ratio of the vector 205 over the scalar VAX was 1225:1. With this speedup ratio, 35 minutes of CYBER 205 time processes the same number of ray-sphere intersections as one CPU month of VAX time.

Table 3. Speedup*.

Primitive	$S_{\frac{205\ vector}{205\ scalar}}$	$S_{\frac{205\ vector}{720}}$
Sphere	33.8	469
Quadric (cylinder, cone, ellipsoid, paraboloid)	16.9	318
Parallelepiped	16.4	298
Torus	10.9	178
Steiner surface	10.7	206

*Speedup $= S_{\frac{P_1}{P_2}} = \frac{\text{CPU time } P_2}{\text{CPU time } P_1}$

The speed increase for the overall program, while not as dramatic as the above comparisons, is certainly of great importance. Three different program/algorithm combinations will be examined: scalar code, vector-intersection code with scalar tree traversal, and the final version of the code containing both vector-intersection and vector-tree traversal code. Figure 3 shows a comparison of the typical percentages of time consumed by major algorithm components for all cases. In this figure the area of a slice is proportional to the actual CPU time it consumed. The missing slice in each pie is miscellaneous overhead. The scalar version of this ray-tracing program typically spent 60 to 75 percent of its time intersecting rays with surfaces and approximately 25 percent traversing the CSG tree. After vectorizing the intersection algorithm, the intersection time fell to 16 to 20 percent of the total CPU time! The CSG tree traversal consumed the same amount of actual time, but the *relative* amount of time it consumed rose from 25 percent to over 60 percent. This is an important result. By taking advantage of the vector processor, the ray-primitive intersection process is no longer the computational bottleneck.

These results initiated an effort to vectorize the CSG traversal routines, the algorithm described in the previous section. The vectorized tree traversal code typically consumes half the CPU time of its scalar counterpart. The smallest circle of Figure 3 shows the vector tree traversal typically requiring 45 percent of the execution time.

Figures 4 and 5 are two scenes for which timing comparisons have been made. The timings presented correspond to scenes computed at a resolution of 1024 by 1024 with one light source and no shadows. Figure 4 shows a model of the Cincinnati-Milacron T3-726 robot composed of 47 primitives: 17 boxes, 23 cylinders, 6 cones, and 1 sphere. The scalar code on the 205 required 1585 CPU seconds to generate this image. With vectorized intersection routines, this time dropped to 548 CPU seconds. Vector intersection and traversal code required 260 seconds. The locomotive shown in Figure 5 took 4852, 1760, and 680 seconds respectively. (The locomotive contains 131 primitives: 41 boxes, 84 cylinders, 2 cones, and 4 spheres.) These two examples indicate an average speedup ratio of approximately 6 by 6:1.

Figures 6 and 7 are other examples of ray-traced images. The self-portrait of the CYBER 205 is a procedurally defined solid. (A closed curve was described in a plane. This plane was swept along its normal to create the volume.) Figure 7 is the locomotive displayed with a distortion. This distortion models the effects of the fisheye lens used in an Omnimax projector. (The work described in this paper was used to create the Purdue section of the SIGGRAPH 84 Omnimax film.)[12]

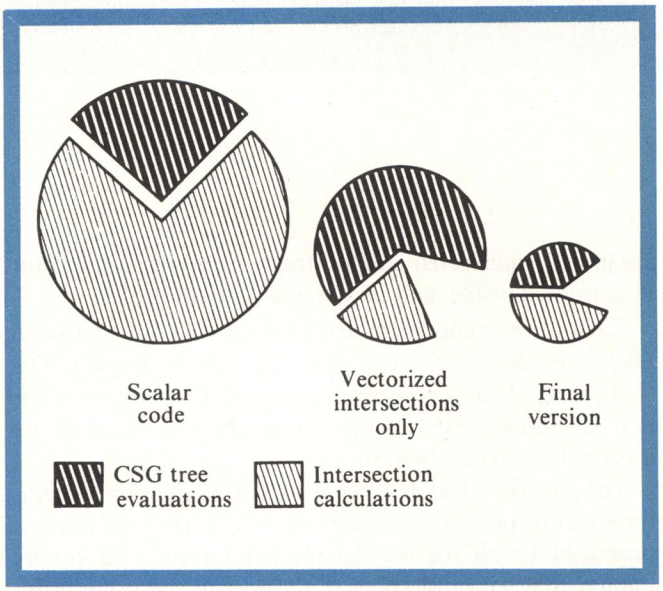


Figure 3. Vector tree traversal.

Conclusions

New technologies in computer architecture are emerging, and one of these, vectorized computing, is tailor made for image synthesis. Exploiting this capability dramatically lowers the execution times for ray-tracing algorithms. Interestingly, the vectorization of the intersection routines resulted in a relocation of the bottleneck of the

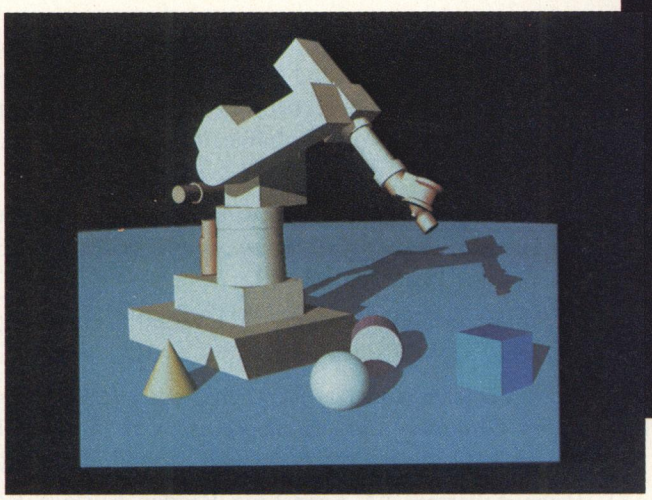


Figure 4. Cincinnati-Milacron T3-726 robot.

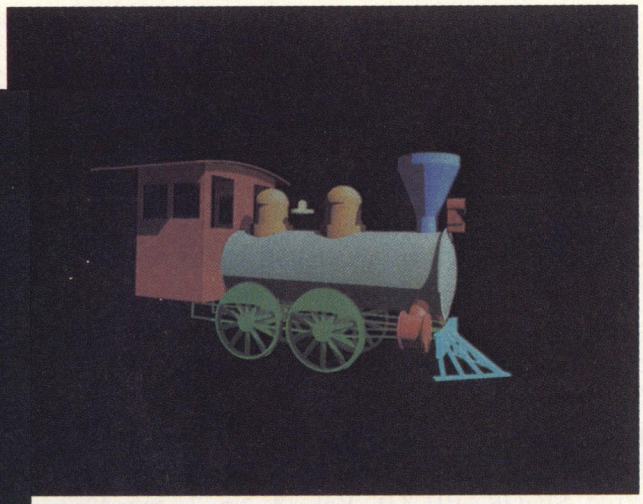


Figure 5. Locomotive.

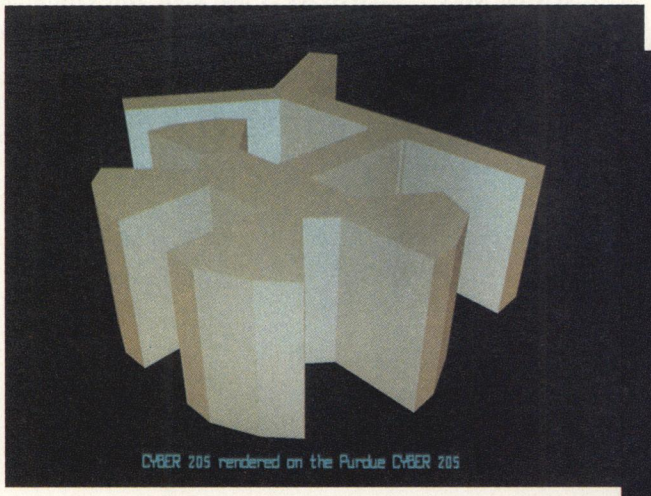


Figure 6. CYBER 205 self-portrait.

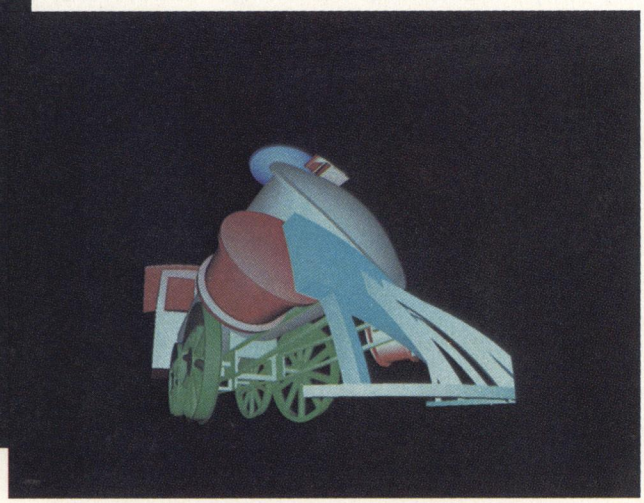


Figure 7. Locomotive with Omnimax distortion.

algorithm from intersection calculations to CSG evaluations.

A noteworthy observation is that, despite requiring more arithmetic operations, the vector code runs significantly faster. This indicates that one of the best ways to exploit vectorization is to homogenize the problem, eliminating as many special cases as possible. This presents an interesting dilemma, which we are still investigating. It seems that we could squeeze more speed if even a gross bounding operation were performed. This needs to be carefully balanced with the need to be as homogenized as possible.

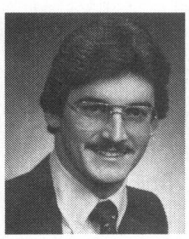
Capabilities such as those available in the CYBER 205 are not an anomaly: They are a trend in computer architecture and thus represent a capability that is becoming more generally available. There are undoubtedly many other computer graphics algorithms that are well suited to take advantage of this type of hardware feature. Scan-line imaging, texture mapping, and antialiasing are three that deserve future attention. ■

Acknowledgments

The authors thank Control Data Corporation for supporting this work under Grant #81P04. The advice and scrutiny of the several reviewers is greatly appreciated. The authors also acknowledge the significant contributions of CADLAB members Bill Charlesworth, Joe Cychosz, John Jackson, and Warren Waggenspack.

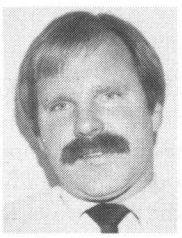
References

1. Roy Hall and Donald Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications,* Vol. 3, No. 8, Nov. 1983, pp. 10-20.
2. James Kajiya, "Ray Tracing Parametric Patches," *Computer Graphics* (Proc. SIGGRAPH 82), Vol. 16, No. 3, July 1982, pp. 245-254.
3. Patrick Hanrahan, "Ray Tracing Algebraic Surfaces," *Computer Graphics* (Proc. SIGGRAPH 83), Vol. 17, No. 3, July 1983, pp. 83-90.
4. James Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics* (Proc. SIGGRAPH 83), Vol. 17, No. 3, July 1983, pp. 91-102.
5. J. Davis, M. Bailey, and D. Anderson, "Projecting Realistic Images of Geometric Solids," *Computers in Mechanical Engineering,* Vol. 1, No. 1, Aug. 1982, pp. 6-13.
6. S. D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing,* Vol. 18, No. 2, Feb. 1982, pp. 109-144.
7. Greg Maxwell and Victor Goldschmidt, "Application of the Ray Firing Method for Determining Radiation View Factors," Report 0469-1, HL83-35P, Purdue University School of Mechanical Engineering, May 1983.
8. Turner Whitted, "An Improved Illumination Model for Shaded Displays," *Commun. of the ACM,* Vol. 23, No. 6, June 1980, pp. 343-349.
9. E. Goldstein and R. Nagle, "3D Visual Simulation," *Simulation,* Vol. 16, Jan. 1971, pp. 25-31.
10. Herb Voelcker and Ari Requicha, eds., *IEEE Computer Graphics and Applications,* Vol. 4, No. 1, March 1982, the issue was devoted to solid modeling.
11. Peter Atherton, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry," *Computer Graphics* (Proc. SIGGRAPH 83), Vol. 17, No. 3, July 1983, pp. 73-82.
12. Nelson Max, "SIGGRAPH 84 Call for Omnimax Films," *Computer Graphics* (Proc. SIGGRAPH 83), Vol. 17, No. 1, Jan. 1983, pp. 73-76.



David J. Plunkett is a member of the technical staff at Unicad, Inc., in Boulder, Colorado. He works on the development of rendering and device-independent graphics packages. Plunkett received his BS and MS in mechanical engineering from Purdue University in 1982 and 1984 respectively.

Plunkett can be contacted at Unicad, Inc., 1695 38th Street, Boulder, CO 80301.



Michael J. Bailey is the newly appointed director of advanced research at Megatek Corporation. When he did the work for this article, he was an associate professor in Purdue University's School of Mechanical Engineering and associate director of the Computer Aided Design and Graphics Laboratory (CADLAB). Besides teaching courses in computer-aided design and computer graphics, Bailey has been involved in consulting activities and was one of two faculty members conducting research in the CADLAB. Before coming to Purdue, Bailey worked at Sandia National Laboratories in Albuquerque, New Mexico, developing CAD and graphics tools for general mechanism design. His areas of interest include computer-aided geometric modeling, graphical man-machine interfaces, image synthesis, mechanism analysis, and animation.

Bailey received his PhD from Purdue in 1979.

Bailey can be contacted at Megatek Corp., 9645 Scranton Rd., San Diego, CA 92121.