Ray Tracing Polygons using Spatial Subdivision

Andrew Woo

Style! Division, Alias Research Inc.
110 Richmond Street East
Toronto, Ontario
M5C 1P1

1. Abstract

Ray tracing consumes a lot of computational resources to render images. This expense usually lies in the ray-surface intersection tests. If the surfaces were polygonal, then we should be able to apply more polygon-specific optimizations to partially cull intersections. Our ray tracer uses a non-memory intensive, voxel traversal intersection culler to assist in such optimizations.

Keywords: intersection culling, polygon, ray tracing, subdivision, voxel traversal.

2. Introduction

Ray tracing [Appe68] [Gold71] is widely acknowledged as a rendering approach that can produce very realistic and beautiful images [Whit80]. It is also widely known that ray tracing is very expensive computationally. Many intersection culling algorithms have been proposed to reduce this expense. However, such intersection culling algorithms do not take into consideration the nature of the primitives which they are culling.

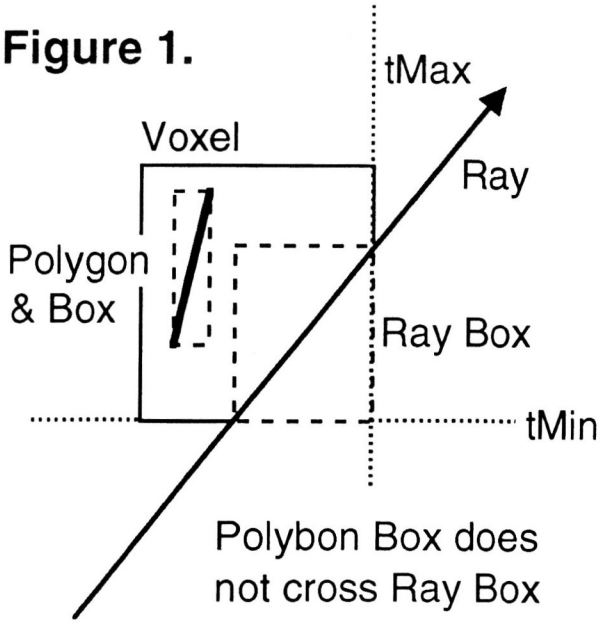
The polygon is one of the most used primitives in rendering surfaces - either as a result of tessellation of complex surfaces, or as descriptions of truncated planar surfaces. In this paper, we examine the polygon very carefully, in hopes of optimizing and reducing the need for ray-polygon intersections, while keeping the memory requirements down to a minimum. The intersection culling algorithm used to assist in such optimizations is uniform voxel traversal [Fuji86]. Our traversal implementation is taken from [Aman87] [Clea88] with ray bounding box checks [Snyd87].

3. Voxel Traveral and Ray Bounding Boxes

A popular intersection culling algorithm is voxel traversal [Glas84] [Fuji86]. Space encompassing all polygons in the scene is divided up into small 3-dimensional boxes, commonly known as *voxels*. Each voxel contains a pointer to polygons that reside in the space occupied by that voxel. Each ray generated traverses the voxel structure in-order and tests for intersection only with polygons residing in voxels that the ray pierces. Thus we hope a small candidate subset of polygons needs to be tested for intersection.

To further reduce the number of polygons that needs to be tested for intersection, Snyder and Barr [Snyd87] proposed the ray bounding box. A ray bounding box in a voxel is created from the ray segment that resides inside that voxel, bounded by the *tMin* and *tMax* extents/distances through the voxel. For each polygon in the voxel, if its bounding box does not cross the ray bounding box, then no intersection test with that polygon is necessary; see figure 1. If they do cross, then the ray-polygon intersection test is needed. This box-crossing test requires at most 6 floating point comparisons.

Figure 1.



This ray bounding box optimization has proven to accelerate the ray tracing culling process by a great deal, especially for densely populated regions distributed in a non-

uniform spatial manner, in which the raw uniform voxel traversal scheme does quite miserably. See table 1 benchmarks for this evidence. However, by using ray bounding boxes, we are restricted to floating point voxel traversal schemes [Glas84] [Aman87] [Snyd87] because the *tMin* and *tMax* values need to be computed, and thus integer-only versions [Fuji86] [Clea88] cannot be used (their increase in computational speed is neglible compared to the advantages of the ray bounding box anyway).

4. Usual Ray-Polygon Intersection Process

The usual ray-polygon intersection test involves the following steps: (1) intersection against the plane on which the polygon lies to compute the hit distance t; (2) check that t is in front of the ray origin ($t > 0$) and t is not in front of any already intersected hits ($t < tHit$) - if either is false, then do not proceed any further as we have already decided that this polygon cannot be the closest visible polygon; (3) use the t value to compute the intersection point; (4) check that the intersection point lies inside the polygon: this is known as the inside-outside check.

Of all the above steps, the inside-outside check (4) is usually the most expensive. So we try to avoid this step as much as possible. One previous attempt to avoid (4) was illustrated in [Woo90], where after step (3), the intersection point is checked against the bounding box of the polygon. If the intersection point lies outside the box, then this polygon cannot possibly be hit by the ray - this check requires 6 floating point comparisons. Furthermore, there is no need to compute all the x,y,z intersection points before checking with the bounding box. Computing the x intersection point followed by checking with the x extents of the bounding box, then repeat with the y and z extents, will be all that much more efficient. This optimization appears to be very effective for tessellated polygons.

5. Order of Candidates for Intersection

In step (2), we also make sure that the t value of the current polygon is not in front of any intersection hits *tHit* that have already taken place, i.e., $t < tHit$. If $t > tHit$, then even if this ray does intersect the polygon, it will not be the closest visible polygon. So why bother with steps (3) and (4)? This leads us to think that it is advantageous to have the closest visible polygon tested for intersection near the beginning and all other candidate polygons can be trivially dismissed from the $t < tHit$ check (as well as the advantage to be described in section 6).

5.1. Dynamic Updating of the Database

For each voxel, there exists a linked list of candidate polygons that occupy the voxel. Ray-polygon intersection tests occur in-order through the linked list. For our optimization, when a ray intersects the closest visible polygon inside the voxel, that polygon is shifted up to the beginning of the linked list. Future rays that pierce the voxel may have the same visible polygon but now have the advantage of intersecting the visible polygon first (or close to first). Then many other ray-polygon intersection tests are rejected at step (2).

This optimization should theoretically be quite effective for shadow determination. It is common practice to assume that what the previous shadow ray hits may be true for the current shadow ray [Hain86]. In addition, shadow rays only need to determine if there exists an intersection hit or not. Thus updating the voxel linked lists might lead to an earlier intersection hit for future neighbouring shadow rays. In addition, this is better than just keeping one polygon pointer to what was previously intersected [Hain86], since neighbouring shadow rays might intersect polygons A, then B, then A, then B, etc. in that order. With our optimization, we should detect intersection within the first few tests of that voxel. In addition, this optimization would be a nice complement to another shadow culler [Pear91] in which triangles residing in the last shadow ray hit voxel are intersected first.

However, we found that the linked list updates for shadow rays do not perform that well. Perhaps the [Hain86] [Pear91] optimizations have already done quite a lot of the work in our implementation already. And having many lights, such linked list updates may prove to be quite useless. Thus, this linked list update is only done for non-shadow rays.

5.2. Almost Hit Cases

In the previous subsection, we only shifted the hit cases up to the front of the voxel's linked list so that future rays will hit the visible polygon near the beginning. However, chances for the same visible polygon are not as likely due to the small tessellated polygons. Thus, we should shift pointers up to the front of the linked list for nearly hit polygons as well. Then we will be able to get even better results on reaching future visible polygons faster.

An almost hit case will be one that returns a *no hit* intersection while in step (4) of the ray-polygon process (inside-outside check), and after the [Woo90] optimization check. This optimization check is quite reliable because it eliminates many candidate polygons, except ones that the ray is close to.

5.3. Using the RayID Effectively

In the previous subsections, we needed to do some linked list shifting. The *rayID* [Aman87] (first proposed to eliminate multiple intersections with the same object in a voxel traversal environment) can be used to select better candidates for intersection without such shifting. For a linked list of candidate polygons to intersect, a 2-pass walk through of the list is needed. The first time the linked list is traversed, only the polygons whose *rayID* is quite close to the current ray's *rayID* are intersected. Then the second time the list is traversed, the remainder of the candidate polygons are intersected.

We do this optimization because the closer the polygon *rayID* is to the current ray's *rayID*, the indication that previous rays have attempted to intersect with this polygon. Thus it is more probable that this polygon may be hit by the current

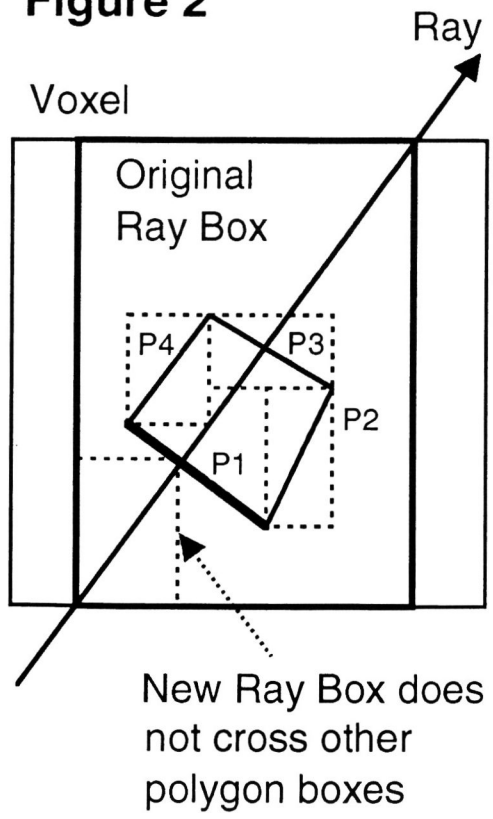
ray. A polygon's *rayID* that is very different from the ray's *rayID* indicates that the polygon has not been intersected lately by previous rays - thus it is likely that the polygon will not be the visible polygon.

On trying out this optimization, it appeared that the *rayID* is not really a good indicator of better candidates, especially when in section 6, dynamic ray boxes are used. Thus, this optimization was removed from our implementation.

6. Dynamic Ray Bounding Box

For each voxel, Snyder and Barr [Snyd87] suggested a box-crossing test between the ray bounding box and each polygon's bounding box. If they do not cross, then we know that intersection with the polygon must fail without any actual ray-polygon intersection tests. We can do a little better: once we get any intersection *tHit* (not just the closest one) with the ray, we can also dynamically reduce the size of the ray bounding box. In other words, the ray bounding box is bounded by [*tMin*, *min*(*tHit*,*tMax*)], instead of [*tMin*, *tMax*] †. With the section 5 optimizations, we hope that the closest visible polygon with hit *tHit* will be encountered near the beginning of the voxel's linked list. As a result of this, the ray bounding box can be adjusted earlier and more ray-polygon intersection tests can be avoided from the box-crossing test.

Figure 2



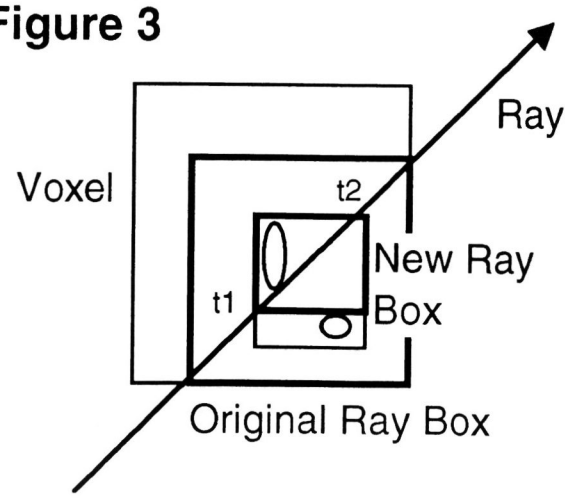
† If *tHit* > *tMax*, then the hit intersection point resides in a subsequent voxel. That voxel, when reached, can be bounded by [*tMin*, *tHit*].

This new ray bounding box does not require additional computation, since it is already done in the ray-polygon intersection hit. In addition, note that this optimzation does not delay intersection, but is a guaranteed culling step; see next subsection's pseudo code. It can be interpreted as a form of object coherence, where we base our assumptions of visibility on a previous neighbouring ray. For example, in figure 2, if we intersect the visible polygon P1 first as a result of the section 5 optimizations, then all the other polygons P2, P3, P4, (which may belong to the same convex surface) residing in the same voxel can be trivially rejected for intersection due to the newly adjusted ray box. If the ray bounding box is not adjusted, then intersections with all those polygons would be needed.

Note that the starting ray bounding box for a voxel does not need to be bounded by [*tMin*, *tMax*] either. For each occupied voxel traversed, we added a ray-box intersection check with the bounding box containing all the polygons (bounded by the voxel) inside the current voxel - refer to this bounding box as *B*. This is done in hopes that the polygons only occupy a small region of the space inside the voxel, and that the ray might miss all of them - we only do this intersection check if *B* is smaller than the voxel itself. If the ray does not intersect *B*, then we just continue traversal. However, if the ray does intersect *B*, we have to perform box crossing and possibly intersection tests with those polygons inside the voxel. Then if the ray intersects *B* at distances t_1, t_2, we can start off with a smaller ray bounding box bounded by [t_1, t_2], where $tMin \le t_1 < t_2 \le tMax$. Note, in figure 3, with this new ray box formed by t_1 and t_2, we do not have to intersect against the bottom sphere.

In the case of second generation rays, it is usually that t_1, $tMin < 0$. Then we bound the ray bounding box by [0, t_2]. This usually avoids the intersected surface, if convex, to test for reflection intersections against itself. Self-mirror reflections cannot occur for convex surfaces.

Figure 3



6.1. Some Pseudo Code

```
/* Standard ray bounding box scheme for each voxel */
rayBox.bound1 = O + voxel->tMin * D;
rayBox.bound2 = O + voxel->tMax * D;
for (each polygon in voxel)
    {
    if (boxesCross (rayBox, polygonBox))
        hit = intersectPoly (&tHit,
            &intersectPoint, polygon);
    }
```

```
/* New dynamic ray bounding box scheme */
rayBox.bound1 = O + max[0,t1] * D;
rayBox.bound2 = O + t2 * D;
for (each polygon in voxel)
    {
    if (boxesCross (rayBox, polygonBox))
        hit = intersectPoly (&tHit,
            &intersectPoint, polygon);

    /* section 5 optimization */
    if (hit || almostHit)
        place polygon at the front of
        the voxel list;

    /* section 6 optimization */
    if (hit && tHit < t2)
        rayBox.bound2 = intersectPoint;
    }
```

6.2. Avoiding Ray Bounding Box Evaluations

In the above pseudo code, the *boxesCross* routine takes at most 6 floating point comparisons. However, this is needed for each polygon in a voxel. In some instances, these ray bounding box evaluations are known ahead of time to not help speed up the ray tracing process, then we might as well do the ray-polygon intersection immediately (without any *boxesCross* tests). This is when the ray crosses a voxel that results in a ray bounding box close to the size of the voxel. We can detect this situation by evaluating $t_2 - t_1 > tMaxVoxel$ for each voxel. If true, then we do not bother with the ray bounding box checks, where $tMaxVoxel = \beta \sqrt{X^2 + Y^2 + Z^2}$ (which is a constant), X,Y,Z represent the dimensions of each voxel, and β is some value close to and less than 1 (a good choice for β is 0.75). Note that if β is 1, then *tMaxVoxel* is just the maximum ray segment that can pass through the voxel.

Then we avoid the box crossing test for this voxel unless the ray bounding box can be reduced by the section 5 and 6 optimizations.

6.3. Multiple Ray Bounding Boxes

If there are a large number of polygons in the voxel, an obvious optimization that can be done here is to have multiple ray bounding boxes for this voxel. Then we have a step-case of smaller ray bounding boxes. If there are r ray bounding boxes, then the voxel's linked list of polygons need to be traversed r times, each time their bounding boxes checked against the current ray bounding box.

This was implemented and tested on our raytracer, but found the results to be quite disappointing and thus was not included in the final raytracer code. It appeared that with multiple ray bounding boxes, less ray-polygon intersections took place. However, many ray-polygon intersections were just delayed by the many box crossing tests. Then the box crossing tests dominated the processing time.

7. The Ray-Plane Intersection

The ray-plane intersection (step 1) needs to be optimized to improve the overall ray-polygon intersection test. The usual computation of t with a plane (or polygon) is:

$$t = \frac{d - N \cdot O}{N \cdot D}$$

where the plane equation is defined by $N \cdot P = d$, P is a $<x,y,z>$ variable triplet, N is the surface normal, and the ray (thus the ray-plane intersection point in step 3) is defined by $O + tD$. We will show that the t evaluations should take 6-8 floating point operations under some circumstances.

7.1. For Second Generation Rays

With second generation rays being shot, we can reuse some previous results to save computation. In other words, for a reflection ray, some of the ray-plane computation can be reused from its parent/cast ray.

We will subscript all cast ray information with c and reflected ray information (extendible to refraction and shadow rays as well) with r. We can expand $d - N \cdot O_r$ to $d - N \cdot (O_c + t_c D_c)$, where t_c is the t value for the closest visible polygon in the cast ray. Then we get $d - N \cdot O_c - t_c N \cdot D_c$. As a result, only an additional multiplication and subtraction are needed to compute the numerator of t_r in step (1): we already know the values of t_c, $d_1 = d - N \cdot O_c$ and $d_2 = N \cdot D_c$ from the cast ray. The reflection ray numerator is then simply $d_1 - t_c d_2$.

After some implementation and testing, we found that this optimization is not worth the trouble due to the small number of same cast and reflection ray hits, and due to the complicated information management within our scheme.

7.2. For Cast (First Generation) Rays

For first generation or cast rays, we notice that the numerator of the t evaluation is always a constant for the same triangle (assuming a perspective view). In other words, $d - N \cdot O_c$ is the same througout each triangle. However, it is far too memory consuming to store the numerator for each triangle in order to save dot product evaluations.

We can preprocess and translate the entire database so that the eye origin resides at (0,0,0). Then $N \cdot O_c = 0$ and the numerator of the t evaluation is d for all triangles, without the need for any storage. Thus $t = d / N \cdot D_c$.

There are other advantages to this translation as well. The ray-plane intersection point computation is simplified from $O_c + tD_c$ to just tD_c. We can use this tD_c to simplify ray bounding box computations for $tMin*D_c$ and $tMax*D_c$ as well. However, we cannot apply this optimization when depth of field effects (along the flavour of distribution ray tracing [Cook84]) are to be generated.

7.3. Dynamic Clipping of t Values

In step (2) when the t value is computed, we need to check that t is in front of the ray origin. In most implementations, a constant fudge value is used to evaluate this: e.g., $t > 0.0001$. However, we can use dynamic clipping of the t extents. We should check that $t > tMin$ instead - actually, it is even better to check that $t > t_1$ (since $t_1 > tMin$).

This optimization allows us to omit steps (3) and (4) if $t < t_1$. This is because the intersection point is far beyond the polygon (beyond the voxel, in fact) that we know it cannot possibly be an intersection hit. With more tight values like t_1 to clip against, the better the chance that steps (3) and (4) can be omitted.

8. Implications of Voxel Subdivision

One open and difficult question in uniform voxel traversal approaches is the subdivision level necessary to get an optimal overall ray tracing performance. If the subdivision level is too deep, then we pay for more traversal and extensive memory costs, but gain the advantage of only needing to deal with small number of surfaces in each voxel. If the subdivision level is not deep enough, we pay for the cost of having to perform more ray-surface intersection tests - this can significantly slow down the ray tracing process. Devillers [Devi88] attempted to answer this question with an analytic solution R for uniformly sized voxels. Subramanian and Fussell [Subr91] also attempted to answer this question in a similar fashion. However, both papers always assumed a $R \times R \times R$ subdivision scheme, instead of a general $X \times Y \times Z$ scheme. In addition, there are so many variables (though spatial distribution and number of polygons play a major role in this analysis) which we must consider that it cannot all be encapsulated in an analytic equation. And is it worth the effort to compute this complicated and expensive solution anyway?

8.1. Previous Benchmarks

With ray bounding boxes [Snyd87], it seems that we are less reliant on the voxel subdivision as compared to the raw voxel traversal approach; the ray bounding boxes act as second level cullers in case many surfaces need to be tested for intersection. And since voxels do take up a lot of memory, it seems that we should consider $X \times Y \times Z$ subdivision schemes that are small. The justification for minimal subdivision can be seen in table 1, where ray bounding boxes (without our optimizations discussed here) were used to accelerate ray tracing on a SUN 3/280 with fpa, and 8 megabytes of memory. The benchmarks were done on a University of Toronto ray tracing program named *optik* with true spheres (not tessellated into many polygons), benchmarked in 1988. The image used was taken from the Haines' sphere flakes image [Hain87], where a densely populated environment distributed in a non-uniform manner is created due to a large floor and a small, concentrated set of spheres. Note that *Ray Box* indicates the CPU minutes taken to ray trace with the ray bounding box and uniform voxel traversal; *Raw Traversal* indicates the CPU minutes taken to ray trace with only uniform voxel traversal.

#Sphere	Grid Res	Image Res	Ray Box	Raw Traversal
7382	40×40×40	512×512	163	391
7382	50×50×50	512×512	142	270
7382	60×60×60	512×512	138	220
7382	70×70×70	512×512	133	199

Table 1: Ray Box vs. Voxel Traversal

8.2. An Approximate Subdivision Level

Another reason we sought a memory conservative voxel subdivision is due to our main platform - the Mac II. Memory conservation is so essential, we can only assume a maximum configuration of 8 megabytes of memory on the machines. This is why many of our optimizations do not take up additional memory and help reduce the usage of memory as well: an alternative to faster but more memory intensive, voxel-based cullers [Jeva89].

We do not try to look for an optimal subdivision level; an approximate one keeping the subdivision minimal should do. Our subdivision scheme to be described below works well in general for polygons. We consider only one main variable: the total number of polygons in the scene - label this n. Let $m = n^{1/3}$, assuming a rather uniform distribution of polygons throughout 3-space. Then we compute the bounding box surrounding all polygons in the scene. Let the spans of the bounding box extents (difference between the maximum and minimum extents) be labelled s_x, s_y and s_z for each of the axes. Then, taking into consideration the spans that occupy the voxel space,

$$X = \frac{s_x}{maxS} m, \quad Y = \frac{s_y}{maxS} m, \quad Z = \frac{s_z}{maxS} m.$$

where $maxS = max(s_x, s_y, s_z)$. This linearity provides us with more cubical voxels than the standard $R \times R \times R$ subdivision. More cubical voxels give a good distribution of polygons within voxels without using up the excess memory imposed by a $R \times R \times R$ subdivision scheme.

8.3. Order Complexities of the Subdivision Level

With this voxel subdivision strategy, we have placed an upper bound on the memory consumption. At worst, the number of voxels is $m \times m \times m$, which actually equals n. And at worst, each voxel will contain pointers to n polygons. Thus the upper bound memory usage is $O(n^2)$ (actually, we can lower this upper bound to $O(n^{5/3})$ for planar polygons if a smart insertion into voxels is done). However, having each voxel containing n polygons is very unrealistic: this means

that all the modelled polygons occupy a large chunk of 3d space. The best case memory usage is O(n), indicating that a polygon resides totally within a fixed number of voxels, which may very well be the norm for tessellated polygons.

Since the complexity of the ray bounding boxes is clearly O(n), then the above paragraph's complexities also hold true for the entire ray tracing intersection culler memory requirements.

9. Testing and Analysis

The testing was done on a standard 68030, 40 MHz, *Mac IIfx* running under the MPW3.2 environment on MultiFinder 6.0.5 with 8 megabytes of memory. The C code was compiled under the standard MPW C compiler. In our implementation, all surfaces are tessellated into triangles and efficiently ray traced in barycentric coordinates. Our raytracer is a ported and re-coded version of Alias' raytracer product, version 3.0. This ported version serves as the raytracer for the Sketch! product on the Mac.

Many previous papers on intersection culling algorithms appeared only interested in the number of ray-surface intersections that were done. Their only aim was to lower the number of such intersections. However, the computation to avoid such intersections may become even more costly than the actual intersection itself. Intersection culling research is at the point now where this number alone has become a less important indicator for speeding up the ray tracing process. Thus, in the upcoming testing sections, the number of intersections done are not stressed in our benchmarks.

A worst-case breakdown of our ray-triangle intersection scheme is as follows: step (1) requires 13 flops, step (2) requires 2 flops, step (3) requires 6 flops, the extra bounding box check [Woo90] requires 6 flops, and step (4) done in barycentric coordinates requires 25 flops. This comes to a total of 52 floating point evaluations per triangle. With the section 7 optimizations, the floating point count for cast ray-intersections is lowered to 43.

9.1. General Testing

The benchmarks in table 2 are listed in total CPU minutes and seconds to render the 640×480 images, and exactly 1 sample per pixel is taken. On average, it appears that we get about 9-14% improvement with our optimizations over the old timings (*New Time* over *Old Time*), where the old timings represent the basic Snyder and Barr culler implementation [Snyd87] and using the subdivision level calculated via section 8's method. Considering how much superior ray bounding boxes are over uniform voxel traversal (as can be seen in table 1), the 9-14% improvement is not too bad. Also note that *%Intersect* is the percentage of ray-polygon intersections saved with our optimizations.

Everything in the scene is made mirror reflective, with a maximum recursive reflection depth of 3. Note also that the lamp image is non-uniformly/sparsely distributed due to the large floor on which the lamp lies on. The lamp image is densely populated due to the small nuts and bolts, as well as the duplicated lamp bowl on the inside and outside.

Image	#Tri	Grid Res	%Intersect	Old Time	New Time
Spheres	3500	15×9×5	13.6%	7:55	6:43
Room	5182	17×17×16	10.2%	37:20	34:01
Lamp	29062	30×10×30	14.9%	25:44	23:27

Table 2: General Optimization Benchmarks

9.2. Object Coherence Testing

We suspect that with a higher sampling rate will come better improvement results for our optimizations. This is mainly due to the assumption that object coherence optimizations in sections 5 and 6 are more likely to get similar hits between subsequent rays (which will provide the majority of the speedups in this paper). The *lamp* image, at resolution 640×480, is being used to test out this assumption (see table 3), where a 6.7% (8.9 – 2.2) improvement jumps to 11.3% (12.2 – 0.9) with more sampling. Note that *sampling* indicates the maximum sampling rate in an adaptive sampling scheme [Whit80], *Tri Time* represents the timings for the optimizations mentioned in section 7, and *New Time* represents the timings for all our optimizations.

Sampling	Old Time	Tri Time	%Improve	New Time	%Improve
1×1	25:44	25:10	2.2%	23:27	8.9%
2×2	37:32	37:11	0.9%	34:07	9.5%
3×3	61:28	60:55	0.9%	54:02	12.2%

Table 3: Lamp Image with Levels of Anti-Aliasing

9.3. Voxel Subdivision Testing

Our voxel subdivision scheme proposed in section 8 needs to be verified, for it is difficult to accept that such minimal subdivision is sufficient in many cases. Most programmers implementing a uniform voxel traversal scheme usually employ a much deeper subdivision level. Table 4 illustrates an example for the spheres image. Note that *space usage* indicates the total memory usage by the program, and note the alarming increase in memory as subdivision level increases for even such a simple scene.

Grid Res	Time	Space Usage
15×9×5	**6:43**	**1,003,852 bytes**
15×15×15	6:44	1,221,752 bytes
20×20×20	6:50	1,331,768 bytes
30×30×30	7:12	1,801,816 bytes

Table 4: Sphere Image with different Subdivision

In addition, note the comparatively small speed difference in table 1 between the different subdivision levels. With our optimizations and having only to ray trace polygons, we expect that the difference will even be narrowed more.

10. Optimization Extensions

The list of optimizations mentioned in this paper can be trivially extended to other surface types as well as culling approaches. For example, the main surface type we considered here is the polygon. However, the use of object

coherence with respect to the ray bounding box (sections 5 and 6) can be applied to other surface types such as general quadrics, parametric and implicit surfaces, etc. In fact, the speedups should be even superior due to the more expensive ray-surface intersection routines for the complex surfaces and the availability of more object coherence (as compared to tiny polygons).

The object coherent ray bounding boxes can be applied to most voxel-based cullers [Glas84] [Aman87] [Synd87] [Jeva89]. Even with hierarchical voxel structures [Glas84], we can apply these optimizations and reduce the voxel subdivision or in case the maximum depth of the voxel is reached but the voxel is still over-populated. Furthermore, if voxels are created on the fly [Jeva89] as needed, as opposed to all preprocessed voxels, then a much higher polygon count should be the limit used before further voxel subdivision is done.

11. Conclusions and Further Discussions

We have listed some simple but rewarding optimizations that can be easily achieved for ray tracing polygons. They have the advantage of requiring neither additional memory nor substantial additional floating point computation. Some of the optimizations can be applied to other surface types and intersection cullers as well.

Another optimization idea is to generate *cartoon reflections* for planar polygons. Such reflections place a decay/fading factor on its intensity, where reflections have no visible effect after a certain maximum distance *maxDist*. A simple decay factor can be $[(maxDist - tHit)/maxDist]^k$; where $tHit < maxDist$. Voxel traversal and ray-polygon intersections will be halted beyond *maxDist*, but approximate reflection information will usually already have been generated. How acceptable and useful is this approximation? Could this decay apply to shadows to fake ambience (shadow intensity as a function of the distance to the closest occluding object) as well? See cartoon reflections image, where the floor has parameters $maxDist = 5$ for reflections, $maxDist = 10$ for shadows, and $k = 1$, i.e. linear decay.

Final thought: if our subdivision level is really quite small and most polygons fit in few voxels, then do we really need the *rayID* concept in this environment? Based on other experiences [Sung91], the *rayID* may not always accelerate the ray tracing process, probably due to extensive memory usage - a 32 bit flag is attached to each polygon. A possible alternative is to make the *rayID* an unsigned short (16 bits), and reinitialize all *rayID*'s after 65535 rays are shot. This was implemented ontop of our raytracer but found no noticeable speedups. Other alternatives are needed...

12. Acknowledgements

Thanks to Andrew Pearce, Steve Chall, Maria Raso (of Alias Research), and Kelvin Sung (University of Illinois) for proofreading and providing valuable suggestions to this paper.

13. References

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

[Appe68] A. Appel, "Some Techniques for Shading Machine Renderings of Solids", Proc. AFIPS JSCC, vol. 32, 1968, pp. 37-45.

[Clea88] J. Cleary, G. Wyvill, "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision", Visual Computer, July 1988, pp. 65-83.

[Cook84] R. Cook, T. Porter, L. Carpenter, "Distributed Ray Tracing", Computer Graphics, 18(3), July 1984, pp. 137-145.

[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 Computer Graphics and Applications, 6(4), April 1986, pp. 16-26.

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

[Gold71] R. Goldstein, R. Nagel, "3-D Visual Simulation", Simulation, January 1971, pp. 25-31.

[Hain86] E. Haines, D. Greenberg, "The Light Buffer: A Shadow Testing Accelerator", IEEE Computer Graphics and Applications, 6(9), September 1986, pp. 6-16.

[Hain87] E. Haines, "A Proposal for Standard Graphics Environment", IEEE Computer Graphics and Applications, 7(5), May 1987, pp. 3-5.

[Jeva89] D. Jevans, B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing", Graphics Interface, June 1989, pp. 164-172.

[Pear91] A. Pearce, D. Jevans, "Exploiting Shadow Coherence in Ray Tracing", Graphics Interface, June 1991, pp. 109-116.

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

[Subr91] K. Subramanian, D. Fussell, "Automatic Termination Criteria for Ray Tracing Hierarchies", Graphics Interface, June 1991, pp. 93-100.

[Sung91] K. Sung, "A DDA Octree Traversal Algorithm for Ray Tracing", Eurographics, September 91, pp. 73-85.

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

[Woo90] A. Woo, "Fast Ray-Polygon Intersection", *Graphics Gems*, ed. A. Glassner, Academic Press, August 1990, pp. 394.

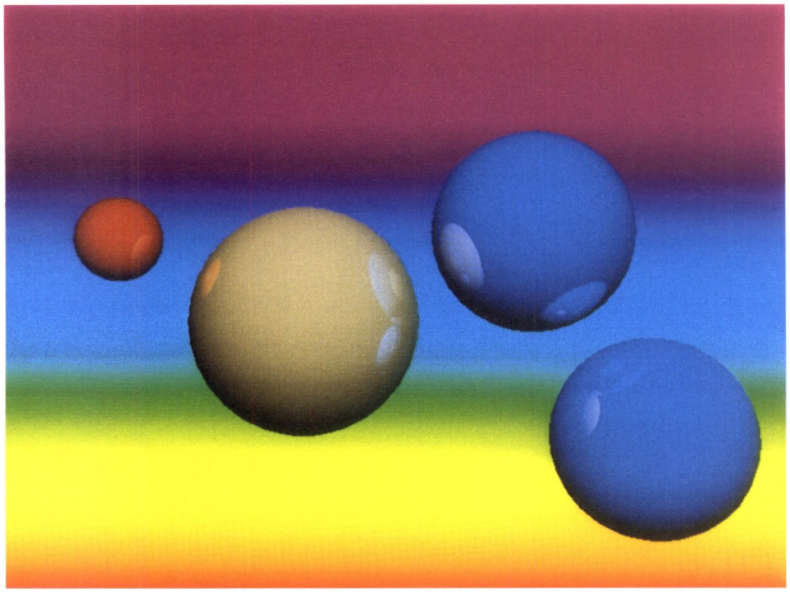


Image 1: Spheres

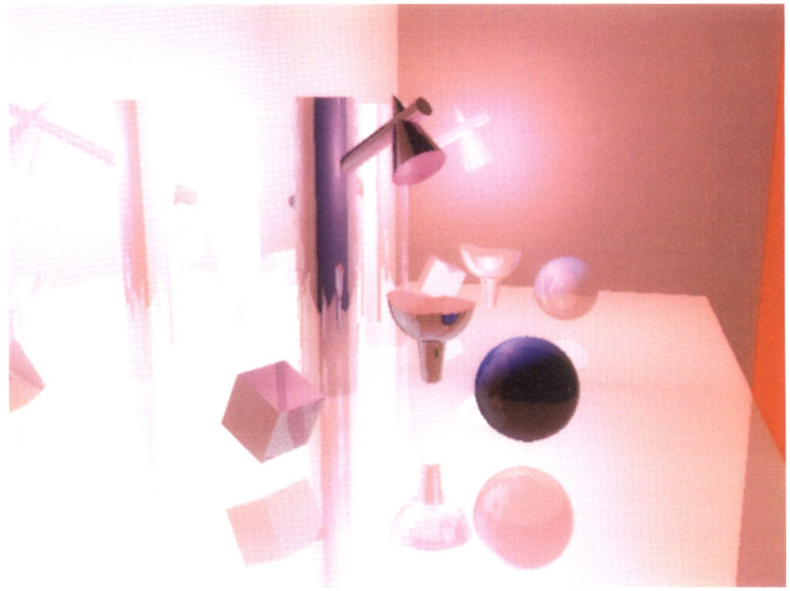


Image 2: Room

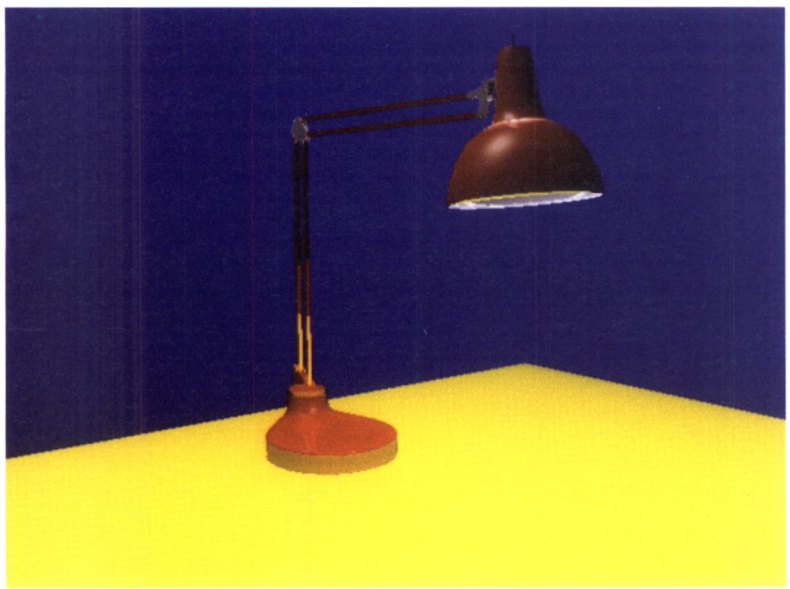


Image 3: Lamp

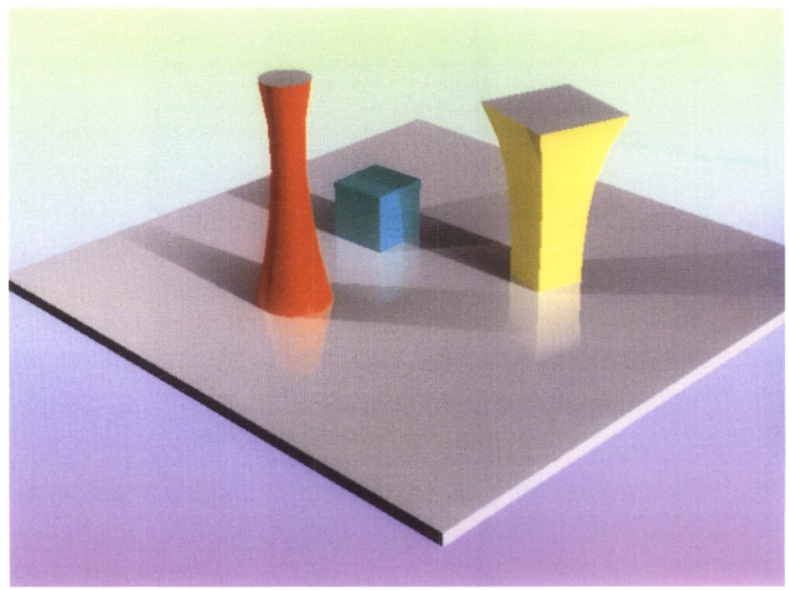


Image 4: Cartoon Reflections