Ray Tracing Triangular Meshes

John Amanatides
Kin Choi

Dept. of Computer Science
York University
North York, Ontario
Canada, M3J 1P3
amana@cs.yorku.ca

ABSTRACT

This paper describes two approaches that allow us to intersect rays with triangular meshes more quickly by amortizing computation over neighbouring triangles. The first approach accomplishes this by performing the in-out test for each triangle using three plane equations, each one representing a boundary edge for the triangle. Each plane is shared between four neighbouring triangles and the cost of intersecting each plane is amortized over all four of these triangles. The second approach, which is more numerically robust than traditional approaches, uses Plücker coordinates to decide on which side of a triangle's edge a ray passes through. In this case side information can be amortized over two triangles. Speedups of over 25% have been recorded when incorporating these approaches in a typical ray tracer. Extending the algorithms to more general polygonal meshes is straightforward.

The Problem

When rendering higher-order surfaces, such as parametric patches, many ray tracers tessellate these surfaces into polygonal meshes and then intersect the resulting polygons (typically triangles) directly [1, 2]. The hope is that the simplicity and speed of the ray-triangle intersection computation will more than offset the fact that many more objects will now have to be intersected.

To get the most performance out of this approach the cost of ray-triangle intersection has to be minimized. Previous solutions have taken approaches that concentrated on dealing with triangles independently. In this paper we develop two algorithms that try to capitalize on the fact that these triangles are part of a mesh and thus try to reduce the total cost of intersecting these triangles.

Previous Solutions

Most graphics systems associate transformations with individual objects in a scene, allowing the user to independently modify each of these objects. When intersecting an isolated arbitrary triangle, the best solution is to precompute a transformation that takes the arbitrary triangle and transforms it to the x-y plane with the vertices mapped to (0, 0, 0), (0, 1, 0), (1, 0, 0). (This transformation is concatenated to the object's transformation and thus requires no extra space or computation.) In this coordinate system computing the ray-triangle intersection is simplified [3]. First, the intersection of the ray and the x-y plane plane is performed (this is simpler than intersecting an arbitrary plane, requiring about 8 flops) and then the intersection point (in 2-D) can be tested to see if it lies within the requisite triangle (only 4 flops).

For each triangle the transformation and its inverse must be stored (21 floating-point numbers total) and every ray must be transformed independently for each triangle (36 flops) even before the actual intersection computation can begin. This solution, which requires 48 flops and is optimal for isolated triangles, needs too much space for polygonal meshes.

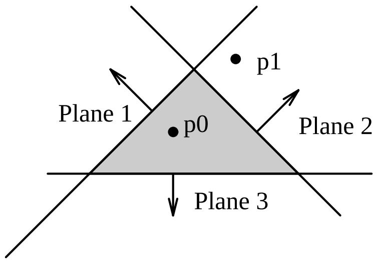
The approach most renderers use when intersecting triangular meshes [2] is to precompute and store a plane that the triangle is embedded in with each triangle and to share the vertex information amongst neighbouring triangles. The ray intersects the triangle's plane (a general plane now) and if the *t* value of the ray's intersection is reasonable, the actual intersection point is computed: $p_t = \text{origin} + t \cdot \vec{dir}$ (21 flops). This intersection point is then tested to see if it is truly inside the triangle; this is typically done using a barycentric approach (approximately 25 flops). The total cost is thus about 46 flops worst case [1, 4]. More recent work by Spackman [5] and Green and Worley [6], by concentrating on determining when the actual intersection point is inside the triangle, have improved on this. Spackman noted that part of the barycentric computation can be precomputed (extra storage is now required per triangle); this reduces the number of flops to determine if there is an intersection to 31 flops worst case. Greene and Worley project the triangle into 2D and store the 2D plane equation for each triangle's edge with the triangle. The intersection point is inserted into these plane equations and reject it if it is on the wrong side. This approach requires 33 flops worst case and also needs extra storage.

Because the above approaches to triangle intersection treat triangles independently there is no way to capitalize on neighbourhood information to reduce the total computational cost. In the next two sections we introduce two such approaches.

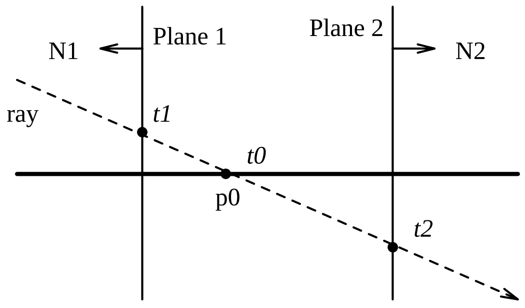
Approach A: Boundary Planes

Consider figure 1a. It shows a triangle embedded in the plane and rather than using vertices, we use three *boundary* planes to define the extent of the triangle. We can now use these planes to decide if a point is inside. In our example, the normals point out. Thus p0 would be inside the triangle (it is on the inside of all the boundary planes) and p1 would be outside (p1 is on the outside of at least one plane).

Using these boundary planes to decide if a point is inside when intersecting triangles (21 flops) is a little faster than the barycentric approach but it requires more space for the planes.



a



b

Figure 1

Consider figure 1b. In this view, the plane that embeds the triangle is perpendicular to the page and we can now see how a ray would intersect the boundary planes (only 2 are shown for clarity). The intersection of the ray and the triangle embedding plane occurs at p0 and has a t-value of *t0*. We can also compute the t-values of the intersection of the ray with the boundary planes and get *t1* and *t2* (for plane 1 and plane 2 respectively).

In deciding if the ray intersects the triangle if we have *t1*, *t0* and *t2* we don't need to compute p0. As the ray is entering plane 1 ($\vec{N1} \cdot \vec{dir} < 0.0$) any t-value greater than or equal to *t1* is OK and since it is leaving plane 2 any t-value less than or equal *t2* is OK. In other words if $t1 \le t0 \le t2$ then the ray intersects the triangle. Thus to determine if a ray intersects a triangle we now need to compute the t-values of the intersection of the ray with 4 planes (the triangle embedding plane and three boundary planes), and compare t-values. This ray classification approach has the flavor of the Cyrus-Beck parametric line clipping algorithm [7].

Below is a procedure that illustrates the approach:

```
Boolean RayIntersectTriangle(Ray *ray, Triangle *triangle,
            double closestHitSoFar, double *tAtHit)
{
    Boolean goingIn;
    double tBasePlane, t;
    int i;

    RayIntersectTrianglePlane(ray, triangle->plane, &tBasePlane);
    if(tBasePlane <= EPSILON || tBasePlane >= closestHitSoFar)
        return MISS;
    for(i=0; i < 3; i++) {
        RayIntersectBoundaryPlane(ray, triangle->boundary[i],
                                    &t, &goingIn);
        if(goingIn){
            if (tBasePlane < t)
                return MISS;
        } else {
            if (tBasePlane > t)
                return MISS;
        }
    }
    *tAtHit = tBasePlane;
    return HIT;
}
```

This approach of comparing t-values, however, is more expensive than using barycentric coordinates and the total cost of intersecting a triangle works out to 63 flops worst case. It also requires space for the planes. However, this is not the end of the story. Consider figure 2.

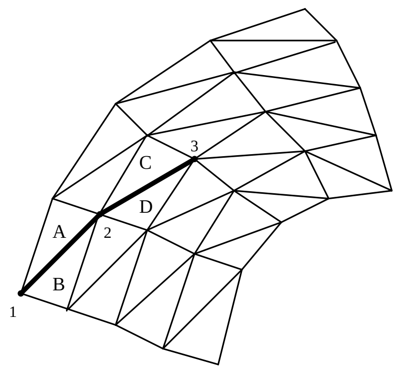


Figure 2

We can construct a plane which goes through the three vertices 1, 2 and 3. This plane will be a boundary plane to triangles A, B, C and D. In a similar manner we can construct triangle boundary planes for all the triangles and each boundary plane (with the exception of the mesh boundaries) will be shared by four triangles.

When we intersect such a boundary plane, the t-value of the intersection as well as whether the ray is going in or out of the plane is identical for all four neighbouring triangles. The cost of the ray-boundary plane intersection can thus be amortized over the four triangles. By sharing this information just under two ray-plane intersections per triangle are required to determine if a ray intersects the triangle. This works out to 29.25 flops worst case and is better than the traditional approaches.

To share this information each triangle points to shared copies of the boundary plane, and the boundary plane data structure is extended to include the shared information:

```
typedef struct {
    Plane p;
    double t;
    Boolean goingIn;
    long rayID;
} BoundaryPlane;
```

Depending on the order in which triangles in the mesh are intersected, it may be necessary to indicate that the boundary plane has already been intersected by the current ray and that the `t` and `goingIn` fields are current. One way of doing this is to give each ray a unique rayID and store the rayID of the ray in the boundary plane whenever it is intersected [8].

One detail that we have glossed over in the above paragraphs is that the normal of a boundary plane that is pointing out for one triangle is pointing in for its neighbour. Consider figure 3:

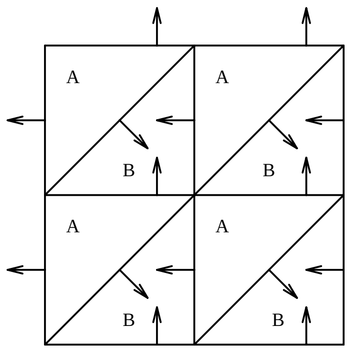


Figure 3

If we consistently orient the boundary planes we find we have two types of triangle in the mesh: type A and type B. Type A triangles have all boundary planes pointing out; type B triangles have all boundary planes pointing in. By replacing one line in the above code fragment we can easily deal with this:

```
    if(goingIn){
```

with:

```
    if(goingIn ^ triangle->typeB){ /* x-or with triangle type */
```

And by keeping two lists in the mesh, one for each triangle type, we can remove the type information from the triangle data structure and reduce the total space requirments.

The above algorithm determines if a ray intersects a triangle and if it does, computes the t-value of the intersection. This information is sufficient for shadow computation but is insufficient for primary rays. In this case, one may need to know the normal, color and texture coordinates at the point of intersection. This information is typically computed by interpolating the values at the verticies using the barycentric coordinates to do the interpolation. When using our algorithm for primary rays this barycentric interpolation computation is only performed for the one visible triangle after we have determined that it is visible; this computation is unnecessary for all the other triangles.

Approach B: Plücker Coordinates

The second approach that we will introduce is based on Plücker coordinates. Plücker coordinates are an alternate way of describing directed lines in three space using six numbers [9, 10]. By peforming a six-dimensional permuted inner product of these numbers (11 flops) we can determine whether two directed lines intersect (the inner product is 0.0) or whether one passes to one side or the other (depending on the sign of the inner product). (A fuller description of this is found in the appendix). These three possibilities are illustrated in figure 4 (after Teller):

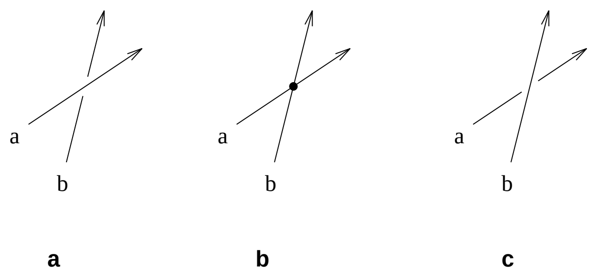


Figure 4

By determining on which side a line passes with respect to another we can determine if a ray passes through a triangle. Consider figure 5:

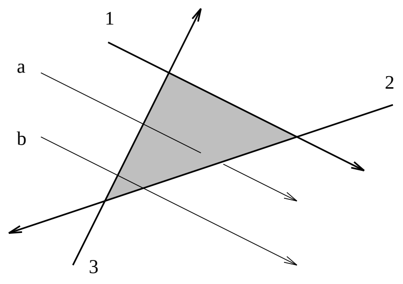


Figure 5

The three lines 1, 2, and 3, help define the edges of the triangle. `A` and `b` are two rays in three space. Ray `a` hits the triangle. Notice that it is on the same side of all three edges. If we perform the Plücker inner product of the ray with each of the lines they will all have the same sign. A ray which doesn't hit the triangle, such as ray `b`, cannot be on the same side of all three edges; one edge will be different from the rest. Now, if we perform an Plücker inner product, one if the edges will have a different sign.

By carefully noting the resulting sign when we do Plücker inner products of the ray with the edges we can now formulate a new algorithm for determining if a ray intersects a triangle. The new procedure requires 51 flops worst case. However, just as we amortized many of the flops in the boundary plane approach we can do the same thing here. Each edge of a triangular mesh is shared by two triangles (with the exception of the boundaries) and the side information is identical for both. Thus by sharing this information we can reduce the number of flops required to do the above computation to 33 flops worst case. To do this we will need to extend the edge data structure (which consists of the six Plücker coordinates) to include both the side information as well as the rayID.†

```
typedef struct {
    float plucker[6];
    long rayID;
    Boolean side;
} Edge;
```

An additional benefit of this approach is that it is numerically robust. Single-precision floating-point arithmetic is

† If space is scarce we need only store three of the Plücker coordinates as the other three can be computed with three subtractions. As well, we can collapse the side and rayID variables into one long by incrementing rayID by two instead of one every time and using the lowest-order bit to store side (we can also do the same thing with rayID and goingIn in the boundary plane approach).

sufficient for both storing the Plücker coordinates and performing the inner product. One does not have to worry about rays falling through the cracks between triangles as an incorrect determination because of round-off of the Plücker coordinate will simply indicate that the neighbouring triangle was intersected.

```
Boolean RayIntersectTriangle(Ray *ray, Triangle *triangle,
            double closestHitSoFar, double *tAtHit)
{
    Boolean side0, side1, side2;
    double t;

    side0= Side(ray->plucker, triangle->edge[0]->plucker) >= 0.0;
    side1= Side(ray->plucker, triangle->edge[1]->plucker) >= 0.0;
    if (side0 != side1)
        return MISS;
    side2= Side(ray->plucker, triangle->edge[2]->plucker) >= 0.0;
    if (side0 != side2)
        return MISS;

    RayIntersectTrianglePlane(ray, triangle->plane, &t);
    if(t <= EPSILON || t >= closestHitSoFar)
        return MISS;
    *tAtHit = t;
    return HIT;
}
```

Note: if the ray is coplanar with the triangle the Plücker inner product returns 0.0 for all three edges and thus would seem to indicate a hit. However, the t-value range test in the second half of the procedure will cull out such coplanar triangles and correctly return a miss.

By carefuly labeling edge directions we can again classify the triangles into two types: A and B. In this approach the type indicates either clockwise or counter-clockwise edge orientations (see figure 6). The above procedure doesn't depend on this orientation and consequently handles both orientations without modification.

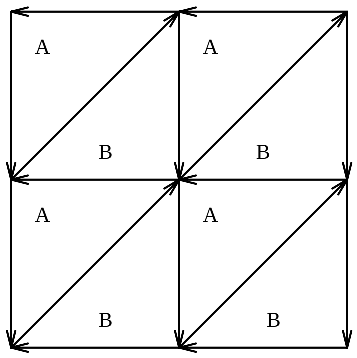


Figure 6

When comparing the two approaches we see that they perform their computations in opposite order: the boundary plane approach does a t-value range test and then checks if the intersection is within the triangle, while the Plücker approach first makes sure that the ray pierces the triangle and only later does it perform the t-value range test.

Note: the amount of extra space required by the edge data structures of the Plücker approach (and the boundary plane approach) is less than either Spackman or Green and Worley's methods.

Implementation and Results

Both approaches were implemented (along with the barycentric approach and those of Spackman and Green and Worley) and tested on a 100 MHz SGI Indigo workstation [11]. Implementation revealed that there were several special cases that had to be addressed, especially in the boundary plane approach. Consider figure 2. If point 1 is coplanar with triangle D then the boundary plane going through points 1, 2 and 3 is the same as the triangle embedding plane for triangle D. In this situation it is now quite possible that both triangles A and B are on the same side of the embedding plane instead of being on opposite sides. In such situations, and in situations where we have degenerate edges (consider the top of a Utah teapot where several control points are coincident) we can only share the boundary plane among two triangles instead of four.

In the first test the scene consisted of a simple pot made up of 24 b-spline patches and one light source. Plate A shows the scene rendered at 256x256x1 using the Plücker approach. Intersection candidate culling was minimal (a bounding box around the whole pot). This allowed us to see how the algorithms compared with each other without intersection culling influencing the results. Below is a summary of the results:

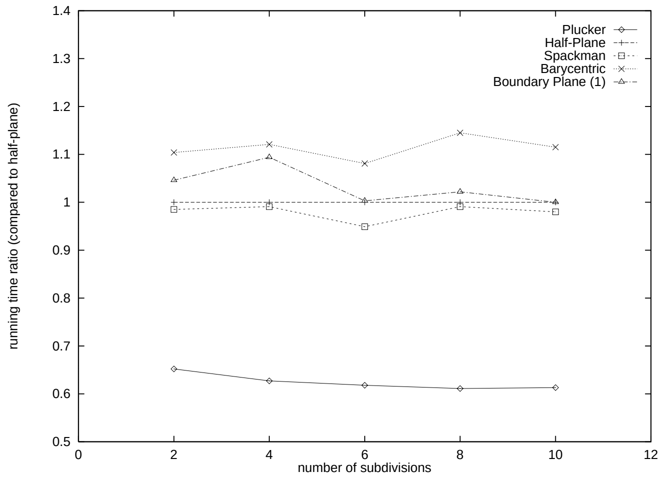


Figure 7

The horizontal axis indicates the level of subdivision of the patches while the vertical axis indicates the ratio of the running time of the new approaches compared to the half-plane approach of Green and Worley. The barycentric and the Spackman approaches were also tested. Note, a subdivision level of n produces $2n^2$ triangles.

As can be seen, the Plücker approach was about 35% faster. An advantage with the Plücker approach is that the first half of the computation evaluates shared information, the Plücker inner products; the `RayIntersectTrianglePlane` computation, which cannot be shared, is performed afterwards. If it is decided early that no intersection is possible the Plücker inner product results can still be shared and amortized. Measurements revealed that over 70% of the intersection computations return miss after the first comparision, with only 2 Plücker inner products computed. This explains why the average case performance of the Plücker approach is much better than the worst case.

The boundary plane approach was not as good, better only than the barycentric approach. The order of computation of the boundary plane approach is such that the shared computation, which can be amortized, is done last; the `RayIntersectTrianglePlane` computation is done first. As well, measurements revealed that less than 25% of the intersections return miss after the first part of the computation (this number would be equally true for all the approaches except Plücker). A variant of the boundary plane approach was created to see if reversing the order of computation might produce better results.

After encorporating a more reasonable intersection culling algorithm, uniform spatial subdivision with a grid of 10x10x10 [8], the following graph was produced:

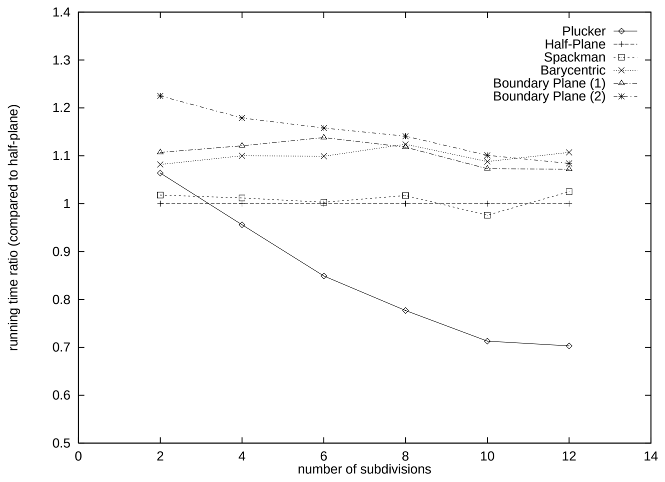


Figure 8

For very low subdivision rates (less than 1 triangle/voxel) the Plücker approach was average and only started to do well when there was a moderate amount of subdivision. At very low subdivision rates only a very small number of triangles are intersected and thus there is little amortization of computation. Once the subdivision rate was reasonable the Plücker approach did well. By the time 50% of the edge compuatations were shared the Plücker approach was 10-15% faster.

The boundary plane approach was uniformly worse. Reversing the order of computation didn't help. More careful measurement revealed that the shared computation did not cull enough rays early enough in the boundary plane computation to make it as big a win as for the Plücker approach. As well the boundary plane approach as it has to amortize over four triangles, not two as in the Plücker approach.

A scene consisting of a Utah teapot (28 Bezier patches) and one light source rendered at 256x256x1 was also tested (plate B) and produced the following:

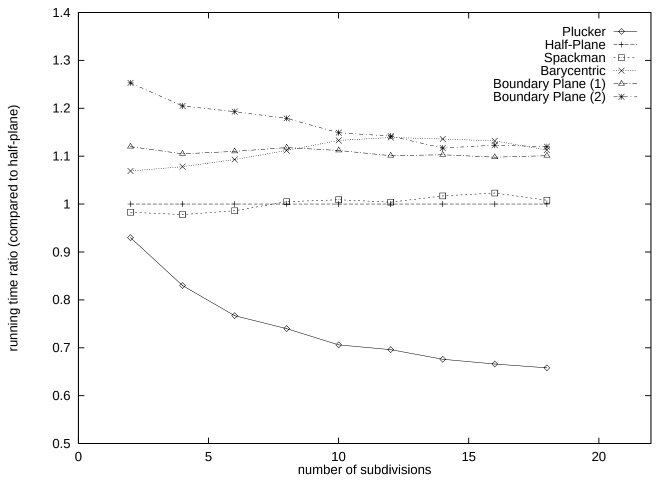


Figure 9

Again the Plücker approach produced superior results. Note: we did not program neigbouring triangles along the borders of Bezier patches to share information so these results can be improved upon.

Conclusions

We have introduced two new approaches for ray tracing triangular meshes. The second approach, using Plücker coordinates, promises significant speedup over current approaches (we have observed speedups of better than 25%), is numerically robust, uses less memory that either Spackman or Green and Worley, and is recommended for general use.

Thanks to Andrew Woo for many useful discussions. Thanks to Pat Hanrahan and James Stewart for pointers to the computational geometry literature and Plücker coordinates. Thanks to NSERC for their financial support.

References

1. J.M. Snyder and A.H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations," *Computer Graphics,* 21(4), pp. 119-128 (July 1987).
2. A. Woo, "Ray Tracing Polygons using Spatial Subdivision," *Graphics Interface '92,* pp. 184-191 (May 1992).
3. P. Hanrahan, "A Survey of Ray-Surface Intersection Algorithms" in *An Introduction to Ray Tracing,* ed. A.S. Glassner, pp. 79-119, Academic Press, San Diego (1989).
4. D. Badouel, "An Efficient Ray-Polygon Intersection" in *Graphics Gems,* ed. A.S. Glassner, pp. 390-393, Academic Press, San Diego (1990).
5. J. Spackman, "Simple, Fast Triangle Intersection, part II," *Ray Tracing News,* 6(2) (July 1993).
6. C. Green and S. Worley, "Simple, Fast Triangle Intersection," *Ray Tracing News,* 6(1) (January 1993).
7. M. Cyrus and J. Beck, "Generalized Two- and Three-Dimensional Clipping," *Computers and Graphics,* 3(1), pp. 23-28 (1978).
8. J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," *Eurographics '87,* pp. 1-10 (August 1987).
9. D.M.Y. Sommerville, *Analytical Geometry of Three dimensions,* Cambridge University Press, Cambridge (1959).
10. S.J. Teller, "Computing the Antipenumbra of an Area Light Source," *Computer Graphics,* 26(2), pp. 139-148 (July 1992).
11. K. Choi, *Ray Tracing Triangular Meshes,* M.Sc. Thesis, Department of Computer Science, York University (1995).

Appendix

(The following is taken from Teller's paper) Consider a directed line going through the points $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$. We can compute the following:

$$\pi_{l0} = p_x q_y - q_x p_y$$

$$\pi_{l1} = p_x q_z - q_x p_z$$

$$\pi_{l2} = p_x - q_x$$

$$\pi_{l3} = p_y q_z - q_y p_z$$

$$\pi_{l4} = p_z - q_z$$

$$\pi_{l5} = q_y - p_y$$

Given two directed lines, a and b, we can take the following permuted inner product to tell us which side one directed line is with respect to another:

$$Side(a, b) = \pi_{a0}\pi_{b4} + \pi_{a1}\pi_{b5} + \pi_{a2}\pi_{b3} + \pi_{a4}\pi_{b0} + \pi_{a5}\pi_{b1} + \pi_{a3}\pi_{b2.}$$

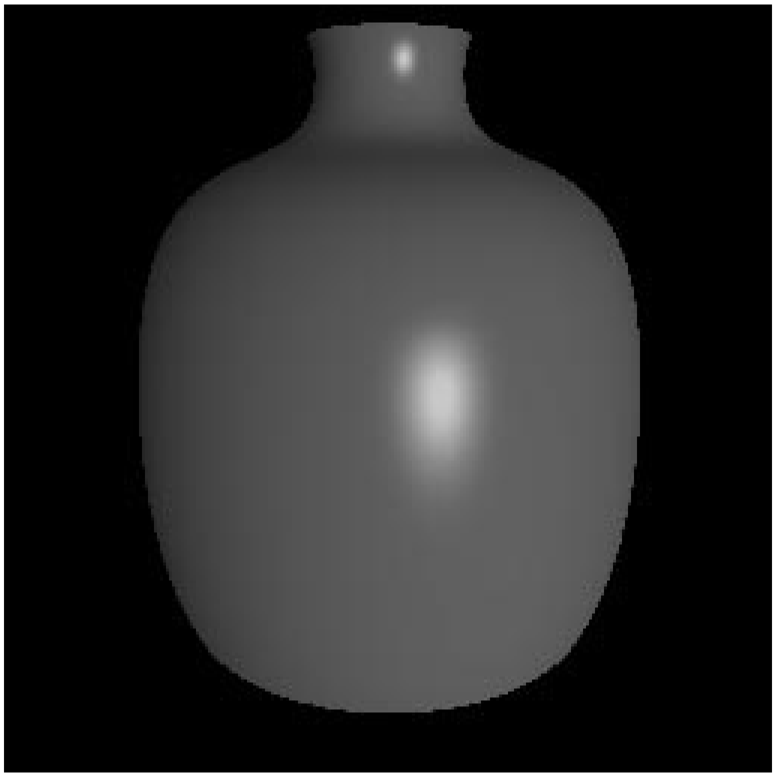


Plate A

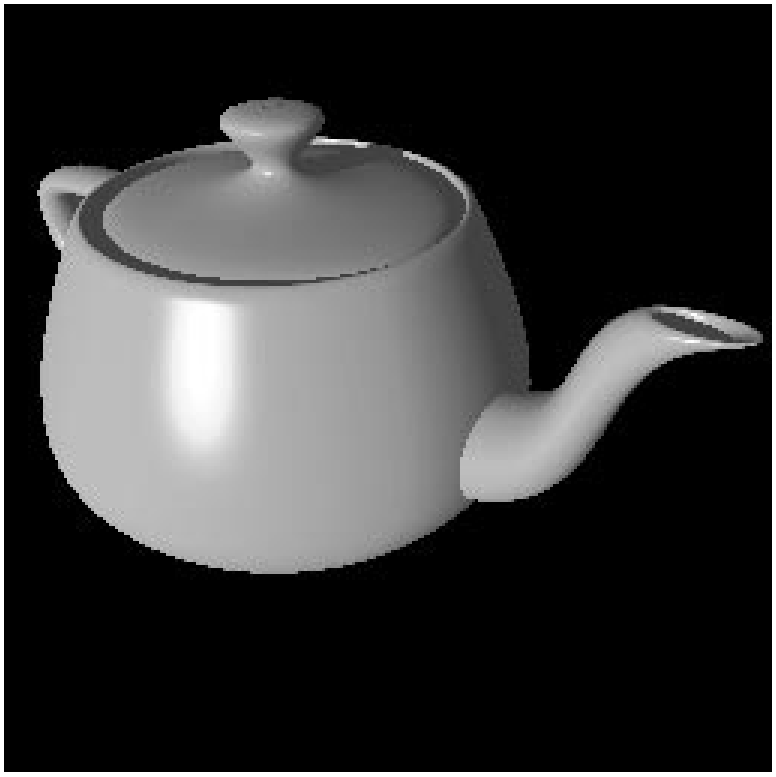


Plate B