Simple Nested Dielectrics in Ray Traced Images

Charles M. Schmidt and Brian Budge
University of Utah

Abstract

This paper presents a simple method for modeling and rendering refractive objects that are nested within each other. The technique allows the use of simpler scene geometry and can even improve rendering time in some images. The algorithm can be easily added into an existing ray tracer and makes no assumptions about the drawing primitives that have been implemented.

1 Introduction

One of the chief advantages of ray tracing is that it provides a mathematically simple method for rendering accurate refractions through dielectrics [3]. However, most dielectric objects are part of a more complex scene and may be nested in other objects. For example, consider an ice cube partially submerged in a glass of water. The ice, water, and glass each have different indices of refraction which would change the ray direction differently. Moreover, the change in ray direction depends on both the refractive index of its current medium and of the medium it is passing into. If a ray was entering ice from water, it would change direction differently than if it was entering ice from air.

A common method for rendering nested dielectrics is to model the scene ensuring that no two objects overlap. This can either be done using constructive solid geometry (CSG) or by manual manipulations of the geometry. A small gap is placed between objects to ensure that rays are never confused about which object they are hitting. This method presents a challenge in that the gap must be large enough so that floating point errors do not transpose the object borders, but if the gap is too large it becomes a visible artifact in the rendered image. Our method allows nested dielectrics without requiring the renderer to support CSG primitives and without needing any gap between the nested objects. It can also allow certain pieces of geometry to be modeled at a lower resolution than would otherwise be necessary. Finally, the algorithm allows some surfaces to be ignored by the renderer reducing the rendering time needed for some models.

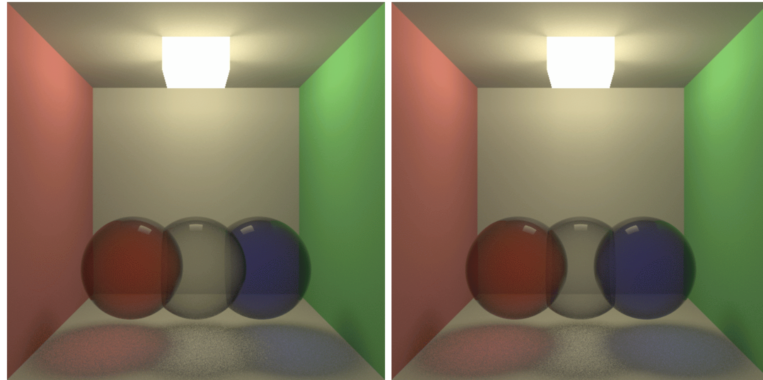


Figure 1: The same scene rendered with different priorities. In the first image, priorities decrease from left to right. In the second, the middle sphere has the lowest priority. To make the difference more visible, the spheres have been given the same refractive indices as the surrounding air.

2 Algorithm

Our method works by enforcing a strict hierarchy of closed geometry. All potentially overlapping materials are represented as closed solids and given a priority when they are defined. Our algorithm works by ensuring that if a ray is traveling through multiple objects, only the object with the highest priority will have any effect on the behavior of the ray. Essentially, the algorithm is a simplified form of CSG applied to the refraction problem in that object interfaces are defined by a geometric difference operation. This operation is controlled by the object priorities. Figure 1 demonstrates a scene using two different sets of priorities.

For our algorithm nested objects should be modeled in such a way that *ensures* they overlap. For example, if rendering a glass filled with water, the boundary of the water would be set between the inner and outer walls of the glass. The modeler would assign a higher priority to the glass to ensure that, when a ray passed through the overlapping region, this region would be treated as part of the glass.

To determine which object a ray is effectively traveling through, the algorithm uses a simple structure called an *interior list*. Interior lists are small arrays stored with each ray that indicate which objects that ray is traveling through. Due to the fact that objects overlap, a ray's interior list may contain multiple objects. The

highest priority object of a ray's interior list is the object which will influence the ray's behavior.

In order to handle the fact that objects overlap, all object intersections are evaluated using the interior list and priority numbers. Since only the highest priority object is considered to exist when multiple objects overlap, we have two cases: the ray intersects an object with a priority greater than or equal to the highest element in the ray's interior list (called a *true intersection*), or the ray intersects an object with a lower priority than this greatest interior list element (called a *false intersection*). Rays with empty interior lists will always produce true intersections. Examples of true and false intersections are shown in figure 2.

This algorithm can be utilized in virtually any ray casting scheme including path tracing [2] and photon mapping [1] and should require only modest modifications to most existing renderers. These modifications are added to keep the interior list updated and to differentiate between true and false intersections.

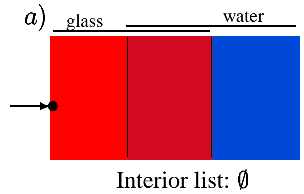
2.1 False Ray Intersections

When a false intersection is encountered no color calculations are performed and we simply continue searching for the next closest intersection. ("Color calculations" refer to the spawning of reflection and refraction rays, lighting, shadowing, and other similar calculations that would contribute to the color discovered by the given ray.) This search is repeated until a true intersection is found or all possible intersections have been shown to be false, the latter indicating the ray missed all geometry.

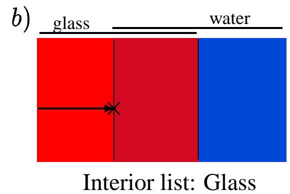
The only computation made as a result of a false intersection is in the interior list. The intersected object is added to or removed from the ray's interior list based on whether the ray entered or exited this object respectively.

2.2 True Ray Intersections

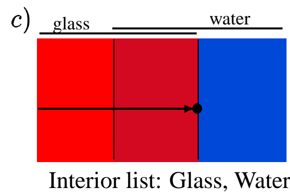
True intersections result in normal color calculations, just as they would in a normal ray tracer. Unlike a standard ray tracer, however, the reflection and refraction rays have interior lists which must be initialized. The reflection ray is simply given a copy of the original ray's interior list since the reflection ray crosses no additional boundaries. The refraction ray, however, is created by crossing from one object to another, and therefore would have a different interior list from the original. The refraction ray starts by copying the interior list of its parent, but then adds or removes the intersected object (depending on whether the refraction ray is entering or exiting this object respectively).



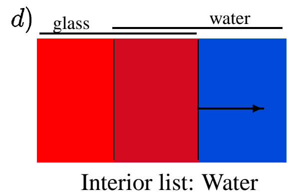
The ray intersects the glass from the outside. Since the ray did not begin in any object (the interior list is empty) this is guaranteed to be a true intersection. We would compute the color values for this point. The reflection ray would continue to use an empty interior list. The refraction ray is shown in b.



The refraction ray from a continues into the glass. It next strikes the border of the water (entering the area where both water and glass are specified). Because the glass has a higher priority than water, the intersection with the water is a false intersection. The interior list is updated and the ray continues to search for an intersection.



The ray next strikes the other side of the glass. Because the glass is equal to the highest priority object in the interior list (itself) this is a true intersection. Color values for this point are calculated. The reflection ray's interior list would contain both the glass and the water objects. The refraction ray is shown in d.



The refraction ray from c continues into the water.

Figure 2: True and false ray intersections. Glass (red) has a higher priority than water (blue). The dark red area indicates where both materials overlap. Note that in a real image the ray direction between a and b and between c and d would likely change due to refraction. (This was not done here to simplify the figure.) There would be no change in direction between b and c since the intersection in b is false.

At the same time, to compute the direction of the refracted ray it is necessary to know the refraction index of the current medium (the “from-index”) and of the medium the ray will be transitioning into (the “to-index”). If the refraction ray is entering the intersected object, the from-index would be the index of the highest priority object in the original ray’s interior list and the to-index would be that of the intersected object. If the refraction ray is exiting the intersected object, the from-index would the the index of the intersected object and the to-index would be the index of the highest priority object in the refraction ray’s interior list. If a ray’s interior list is empty, this indicates that ray is traveling outside all geometry. Usually this space is given the refractive index of air, although any index could be assigned.

3 Discussion

The key contribution of this algorithm is that it allows objects to overlap in model space while still producing correct borders in the rendered image. The method is relatively simple, but still manages to produce strong performance.

3.1 Advantages

This algorithm can significantly simplify the modeling of nested dielectrics. Consider a glass filled with water. Previously, the water would have been modeled as slightly smaller than the inside of the glass. In order to keep the gap between the objects as small as possible the border of the water would need to be rendered at high resolution to closely follow the glass’s surface. Using the method proposed in this paper the sides of the water could be anywhere between the sides of the glass. Since the sides of the water would only be used to mark the water’s boundary and would never be rendered, they could be modeled at a lower resolution. Only the glass boundaries would be modeled at high resolution because only these boundaries would be visible in the rendering.

A second advantage of this method is that it makes the modeling of some surfaces unnecessary. If a single surface forms the boundary between two objects only the higher priority object needs to define this boundary. Consider gas bubbles completely surrounded by water. Previously it would have been necessary to model a border for the water surrounding the bubbles as well as modeling the bubbles themselves. However, using our technique, if one gives the gas bubbles a higher priority than the surrounding water only the bubbles would need to be modeled. As a result, careful ordering of priorities can actually reduce the number of boundaries against which intersection calculations must be performed.

A third advantage is that, because false intersections do not require color calculations, rendering time can actually be reduced in some models. Again, consider a glass with water in it. In a normal ray tracer, color values would be calculated when the ray entered the glass, exited the glass, and entered the water for a total of three calculations. Using our algorithm, one of these intersections would be false and would receive no further computation. As a result, the same set of ray intersections would result only in two color computations.

3.2 Implementation and Limitations

In order to keep track of which objects a ray is inside, our implementation simply toggles interior status of an object each time a ray crosses its boundary. If a ray intersects an object and the object is not in the interior list then the ray must be entering the object. If the intersected object is already in the interior list, the ray is currently traveling through the object's interior and hence would exit the object at this point. This technique fails when the ray has a singular intersection with an object, such as along the object's silhouette. The use of a more sophisticated algorithm for determining which objects an ray was interior to at a given point could most likely eliminate these errors. However, even when using our simple toggle method, we found that standard anti-aliasing techniques removed most artifacts.

The primary disadvantage of the algorithm is the constraint that nested geometry must have overlapping borders. This requirement can provide some added complexity, especially if the surrounding material is very thin. In most cases, however, a simple scaling of the interior object by a small amount will be a sufficient solution.

3.3 Efficiency

Our algorithm requires very little additional overhead. In our implementation, when a false intersection is encountered, we simply moved the base of the ray to the location of the false intersection and re-cast the ray in the same direction. This process is repeated until the nearest intersection is a true intersection. Despite the inefficiency of this method we found that the use of this algorithm had a minimal impact on the overall time of the rendering and could, in some cases, actually reduce the rendering time. Specifically, in scenes with no dielectrics we found the algorithm to be only about 0.8% slower, while scenes with multiply nested refractive objects, such as the glass in Figure 3, could have their overall rendering time reduced by more than 5% compared to an unmodified ray tracer.

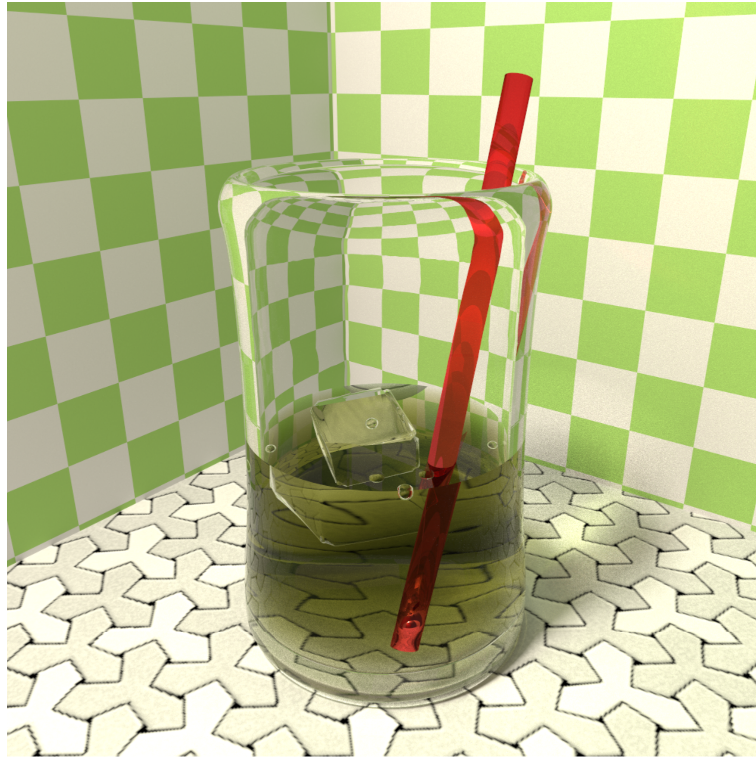


Figure 3: Glass with nested dielectrics. Priorities (from highest to lowest) are: the glass, the ice cube, the air bubbles under the water, and the water.

Our implementation uses a short array of pointers to scene objects as the interior list. Because it is unlikely that a ray will start from inside more than a handful of objects the interior list need only be large enough to contain a few elements. We found that the computations related to maintaining the interior list accounted for less than 0.4% of the runtime of the program, even for scenes with multiply nested objects.

Acknowledgments. Special thanks to Margarita Bratkova for creating the glass and ice cube models. This material is based upon work supported by the National Science Foundation under Grants: 9977218 and 9978099.

References

[1] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[2] Peter Shirley. *Realistic Ray Tracing*. A K Peters, 2000.

[3] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

Web Information:

http://www.acm.org/jgt/papers/SchmidtBudge02

Charles Schmidt, University of Utah, School of Computing, 50 S. Central Campus Dr, Salt Lake City, UT (cms@cs.utah.edu)

Brian Budge, University of Utah, School of Computing, 50 S. Central Campus Dr, Salt Lake City, UT (budge@cs.utah.edu)