Ray Tracing CSG Objects Using Single Hit Intersections

Andrew Kensler

April 10, 2006

1 Introduction

Typical approaches to rendering constructive solid geometry (CSG) require finding all intersections of a line with a primitive and then computing the intersections by examining the intervals. Due to the memory costs to store all of the intersections, and the computational costs needed to compute and order them, this approach can be quite expensive.

Moreover, many modern ray tracers support finding only the single nearest intersection. The approach described here computes intersections with binary CSG objects using this style of intersection. Though it may need to do several of these per sub-object, the usual number needed is quite low. The only limitation of this algorithm is that the sub-objects be closed, non-self-intersecting and have consistently oriented normals.

2 Example: Intersecting a Union

Consider a simple union of two overlapping spheres as shown in Figure 1:

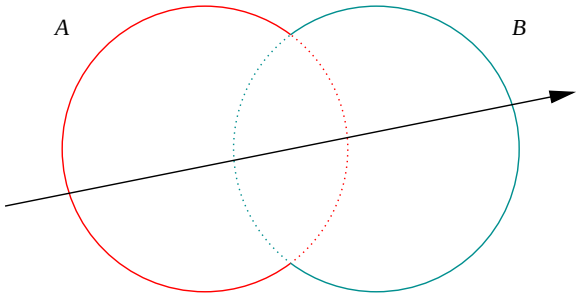

Figure 1: Union of two spheres from outside

The CSG union of two objects is the combined parts of the boundary from each that is not in the interior of the other object. To find the closest intersection of a ray with the union of the two spheres, we need to find the closest intersection of either sphere such that it is not inside the other sphere. In the case above, that would be the first intersection with A.

How can a ray tracer know this? One way is to shoot the ray at each of the sub-objects. Let the ray be defined parametricly as $\vec{O} + t\vec{D}$ and let the nearest intersection with A be at $t = t_A$ where $t_A > min_A$ and the normal at the hitpoint be $\vec{N_A}$, and similarly for B. Initially, min_A and min_B are both 0.

Note that the $\vec{D} \cdot \vec{N_A} < 0$ at the nearest intersection with A. The surface normal points back along the direction of the ray. Since the object is closed and non-self intersecting, this means that O must have been outside A. (Likewise, $\vec{O}$ is also outside B.) Since the intersection with A is the closer of the two, that is the nearest intersection with the union.

Now suppose that we instead shot the ray from inside the union, as shown in Figure 2:

Here the situation is more complicated. The intersection with B is closer than that with A, but $\vec{D} \cdot \vec{N_A} > 0$ is positive, meaning that the intersection with B is inside of A, so we must disallow it. In this case, we can set min_B to t_B and shoot the ray at B again, this time finding the intersection on the far side of B.

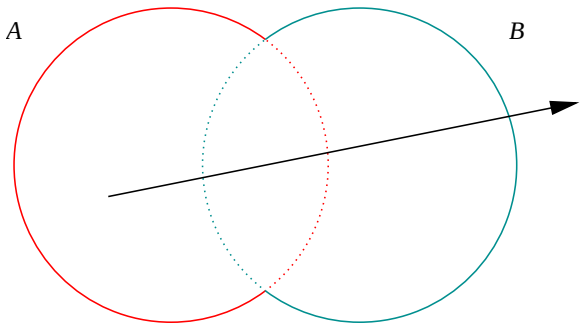

Figure 2: Union of two spheres from inside

The closest intersection is now with the far side of A, but $\vec{D} \cdot \vec{N_B} > 0$ so the intersection with A is on the interior of B and must again be disallowed. So as before, we can set min_A to t_A and shoot again to consider the next intersection with A – this time, no intersection with A will be found.

At this point, there are no more intersections with A, but there is one with the far side of B. Since that intersection can't be inside of A (if it were, we'd still have an intersection with A to consider), the intersection with the far side of B must be the intersection with the union and so we return it.

Of course, if there was no intersection with either A or B, then there can't be an intersection with their union.

3 The State Table

Less abstractly, the approach above could be implemented by starting initializing min_A and min_B to zero, shooting a ray at each subobject to find the intersection with the $t > min$ and classifying the intersections with the subobject as one of entering, exiting or missing it. Based upon the combination of the two, one of several actions might be taken: either returning a hit, return a miss, or set $min = t$ for one of the objects and then shoot a ray and classify it again. If put into a table, the basic actions look like looks like Table 1:

	Enter B	Exit B	Miss B
Enter A	Return A if closer, else return B if closer	Return B if closer, else advance A and loop	Return A
Exit A	Return A if closer, else advance B and loop	Return A if farther, else return B if farther	Return A
Miss A	Return B	Return B	Return miss

Table 1: A State Table

With a few additions to the list of possible actions, the algorithm can be made to work for CSG difference and CSG intersection objects as well. The total list of actions is shown in Table 2:

ReturnMiss	Exit, reporting no intersection
ReturnAIfCloser	Return the intersection with A if it is closer
ReturnAIfFarther	Return the intersection with A if it is farther
ReturnA	Always return the intersection with A
ReturnBIfCloser	Return the intersection with B if it is closer
ReturnBIfFarther	Return the intersection with B if it is farther
ReturnB	Always return the intersection with B
FlipB	If returning an intersection with B, flip its normal
AdvanceAAndLoop	Continue with the next intersection with A
AdvanceBAndLoop	Continue with the next intersection with B

Table 2: Actions

4 Putting it all together

With these, the state tables for CSG union, difference and intersection are shown without derivation in Table 3. Pseudocode for the intersection algorithm to execute these tables is also given below.

Union	Enter B	Exit B	Miss B
Enter A	{ ReturnAIfCloser, ReturnBIfCloser }	{ ReturnBIfCloser, AdvanceAAndLoop }	{ ReturnA }
Exit A	{ ReturnAIfCloser, AdvanceBAndLoop }	{ ReturnAIfFarther, ReturnBIfFarther }	{ ReturnA }
Miss A	{ ReturnB }	{ ReturnB }	{ ReturnMiss }

Difference	Enter B	Exit B	Miss B
Enter A	{ ReturnAIfCloser, AdvanceBAndLoop }	{ ReturnAIfFarther, AdvanceAAndLoop }	{ ReturnA }
Exit A	{ ReturnAIfCloser, ReturnBIfCloser, FlipB }	{ ReturnBIfCloser, FlipB, AdvanceAAndLoop }	{ ReturnA }
Miss A	{ ReturnMiss }	{ ReturnMiss }	{ ReturnMiss }

Intersection	Enter B	Exit B	Miss B
Enter A	{ ReturnAIfFarther, ReturnBIfFarther }	{ ReturnAIfCloser, AdvanceBAndLoop }	{ ReturnMiss }
Exit A	{ ReturnBIfCloser, AdvanceAAndLoop }	{ ReturnAIfCloser, ReturnBIfCloser }	{ ReturnMiss }
Miss A	{ ReturnMiss }	{ ReturnMiss }	{ ReturnMiss }

Table 3: State Tables for CSG Operations

Algorithm 1 Pseudocode for CSG intersection routine

$min_A = 0$
$min_B = 0$
(t_A, $\vec{N_A}$) = IntersectWithA($\vec{O}$, $\vec{D}$, min_A)
(t_B, $\vec{N_B}$) = IntersectWithB($\vec{O}$, $\vec{D}$, min_B)
$state_A$ = ClassifyEnterExitOrMiss(t_A, $\vec{N_A}$)
$state_B$ = ClassifyEnterExitOrMiss(t_B, $\vec{N_B}$)
loop
  $action = table\,[state_A, state_B]$
  if ReturnMiss $\in action$ **then**
    return miss
  else if ReturnA $\in action$ or
      (ReturnAIfCloser $\in action$ and $t_A \leq t_B$) or
      (ReturnAIfFarther $\in action$ and $t_A > t_B$) **then**
    return t_A, $\vec{N_A}$
  else if ReturnB $\in action$ or
      (ReturnBIfCloser $\in action$ and $t_B \leq t_A$) or
      (ReturnBIfFarther $\in action$ and $t_B > t_A$) **then**
    if FlipB $\in action$ **then**
      $N_B = -N_B$
    end if
    return t_B, $\vec{N_B}$
  else if AdvanceAAndLoop $\in action$ **then**
    $min_A = t_A$
    (t_A, $\vec{N_A}$) = IntersectWithA($\vec{O}$, $\vec{D}$, min_A)
    $state_A$ = ClassifyEnterExitOrMiss(t_A, $\vec{N_A}$)
  else if AdvanceBAndLoop $\in action$ **then**
    $min_B = t_B$
    (t_B, $\vec{N_B}$) = IntersectWithB($\vec{O}$, $\vec{D}$, min_B)
    $state_B$ = ClassifyEnterExitOrMiss(t_B, $\vec{N_B}$)
  end if
end loop