Ray-Traced Constructive Solid Geometry

Audrey Baxter, Lorelei Wright
Rensselaer Polytechnic Institute

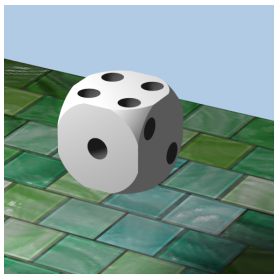

Figure 1: A simple die created as a test scene

Abstract

This project presents an implementation of constructive solid geometry (CSG) for ray-traced rendering. Our first challenge was extending the existing code base to allow for parsing of constructive solid geometry trees. Then, we extended the intersection testing to record spans of intersection with primitives. Finally, we implemented the union, intersection, and difference set operations on collections of spans to resolve the intersection of rays cast through constructive solid geometry objects, and identify the true surface of the object for rendering. Our implementation supports reflection, including self-reflection, and can assign different materials for each primitive in the constructive solid geometry tree.

Keywords: ray-tracing, ray-casting, constructive solid geometry, CSG.

1 Introduction

Constructive solid geometry (Requicha 1977) is a method of modeling which uses the three [1] binary set operations *union* ($S \cup T$), *intersection* ($S \cap T$), and *difference* ($S - T$) to compose primitive objects such as spheres, cubes, and cylinders into shapes which would be prohibitively complex to model purely with implicit geometry. CSG allows for modular composition, that is, CSGs may reference and be composed of other CSGs, building a structure most conveniently represented by a tree such as the one below.

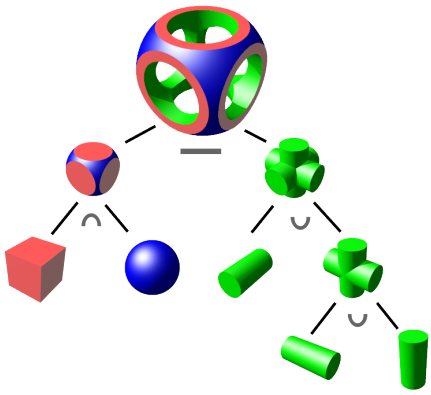

Figure 2: A CSG model with its associated tree

Constructive solid geometry is a form of *solid modeling*, wherein structures are described in terms of volumes rather than surfaces. This means that any arbitrary point in space can be categorized as inside or outside of the object. For our use case, this is an advantage over describing complex models as meshes because significantly fewer rays have to be cast to determine the final intersection point.

[1] A complement operator ($\neg S$) is not necessary, as it can be represented using existing operators with DeMorgan's Law. A similar process can be used to eliminate difference (Duff 1992), but we found difference too practically useful to leave out.

2 Related Work

Constructive solid geometry is only one of several ways to represent solid objects. Others include boundary representations (b-reps), sweeping, parametric surfaces, cell decomposition, spacial occupancy enumeration (SOE), octrees, and binary space partitioning trees (Foley et al. 1990).

In the arena of constructive solid geometry in particular, most advancements are in adjacent territory due to the relatively long-standing nature of ray-traced solutions to CSG rendering:

- Wyvill et al present an extension of CSG with several more operators – including blending, tapering, and twisting – which allow for greater expressivity in models (Wyvill, Guy, and Galin n.d.).
- Chen and Tucker support "blobby" models by representing objects as a continuous scalar field in $\mathbb{R}^3$ rather than as a binary classification, and also utilize volumetric textures for materials (Chen and Tucker 2000).
- Many authors have proposed several different techniques of *boundary evaluation*, or generating a triangle mesh from a solid model. This is a complex problem with active research efforts due to the difficulties in balancing speed and fidelity. (Zhou et al. 2016).

3 Implementation

Constructive solid geometry has been proven extremely well-suited to ray-tracing, and a general solution to CSG-ray intersections was introduced in the very first paper to use the term "ray-casting" (Roth 1982). This project implements this classic solution using C++ and an OpenGL frontend, building off a simple ray tracing engine written for an assignment earlier in the semester.

3.1 OBJ Extensions

That earlier assignment made several additions to the stock OBJ spec to make it more suitable as a scene representation, including an ad-hoc material system, an embedded camera and background color, and syntax for several types of implicit surfaces. This project continued this by adding a handful of new extensions. The first addition was the inclusion of an axis-aligned box primitive, defined by its center and extents. For example, the line `b 0 1 0 5 2 2` defines a box at (0, 1, 0) with a width of 5 and height and depth of 2.

Next, a keyword was added for each operator: `diff`, `union`, and `intersection`. These keywords are called with prefix notation and take two objects (either primitives or a nested CSG object) as arguments. Whitespace is ignored, but is suggested to keep track of the hierarchy involved. It also proved advantageous to support comment lines, so as to not lose oneself in a complicated tree.

```
diff
  union
    #spheres
    s 0 0 0 1.5
    s 0 2 0 1.5
  b 0 1 0 5 2 2
```

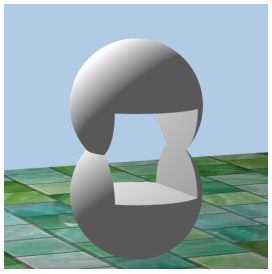

Figure 3: An example CSG tree with its rendering

CSG objects are parsed using a stack structure, pushing new branches when a new operator is encountered and popping operators once both of its children have been parsed.

3.2 Spans

In order to extend the ray tracer to cast rays through CSG objects, we have to know what regions are inside the primitives and which are outside. To do this with ray tracing, instead of keeping track of only the first point where the ray hits the object, we also track the point where the ray exits the primitive. This allows us to store regions along the ray that are inside the primitive. We call these spans.

For the spans, we use the existing `Hit` class, which stores the position, normal, and material of the object at the point of intersection. The `Span` structure

is a POD (plain old data type), simply storing one Hit for when the ray enters the volume, and another when it exits. This gives us not only the segment of the ray inside the volume, but also the surface information (normal and material) at the intersection point. This will be important later on when handling the application of the set operations on collections of sets.

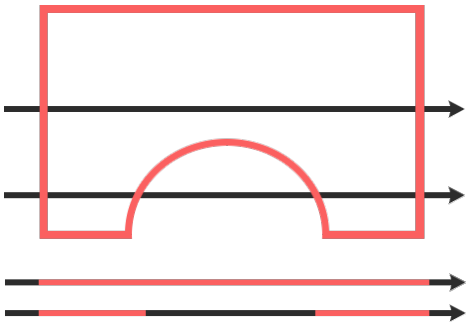

Figure 4: Spans of each ray are highlighted in red

Although a query to all implemented primitives can returns a maximum of one span, a query to a CSG may return one, several, or zero spans. To account for this, spans are accumulated in a collection as they are found, namely a sorted list. As control flows back up the tree after the span of each primitive is known, this list gets modified and merged with other lists as needed to resolve overlap or fracture of spans as a result of the boolean operations.

Once this list of spans has flowed all the way back to the tree's root, we can determine the final, singular intersection point between the ray and the geometry. By storing the list of spans in sorted order, sorted by how close the entering hit of the span is to the camera, we can easily find the nearest span. Once all the set operations have been applied to all of our spans, we simply return the closest hit, entrance or exit, as the final intersection. (Note that spans behind the camera are ignored.) If this is a camera ray rather than a shadow ray, this hit is used to shade the sample produced by that ray.

3.3 Interval Operations With Spans

Recall that the set operations are applied to the spans at the level of rays rather than to the volume itself. The logic for each of the CSG operations is as follows:

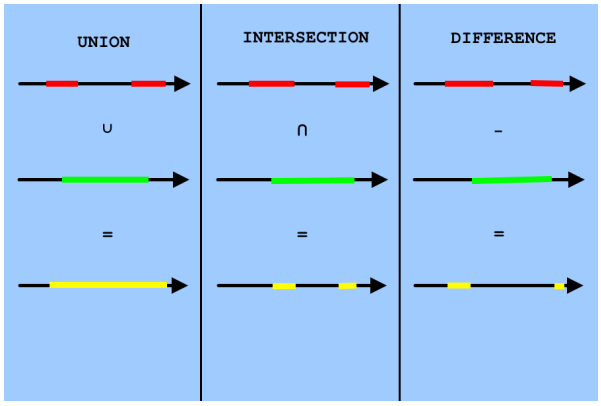

Figure 5: Interval operations on ray-cast spans

- *Union*: If the two spans overlap, return a new span whose closest point is the min of the closest points of the spans being combined, and whose farthest point is the max of the farthest point of the spans being combined. If they do not overlap, return the original spans untouched.
- *Intersection*: If the two spans overlap, return a new span whose closest point is the max of the closest points of the spans being combined, and whose farthest point is the min of the farthest point of the spans being combined. If they do not overlap, return the original spans untouched.
- *Difference*: If both spans have regions not covered by the other, clip the minuend span such that it doesn't overlap the subtrahend span. If the subtrahend span is totally enveloped by the minuend span, return 2 spans connecting each of the minuend. If the minuend span is totally enveloped by the subtrahend span, delete it entirely. No span is returned.

3.4 Materials and Normals

An important consideration when applying set operations is carefully handling normal and material information. The simplest case to handle is the normal case. For union, we don't need to adjust the normal at all, as the closest hit will always be from the first object intersected.

Intersection is trickier, and is the first case where we really had to take care in handling normals and materials. Consider Figure 2. When the sphere and box

are intersected on the left side of the tree, the curvy parts of the new rounded box should be blue like the sphere, and the flat parts should be red like the box. To do this, returning the max hit as described will do the trick, as the hits contain material information, and the maximum distance hit will contain the material information for that primitive.

Difference again is another important place to be weary. In Figure 2 again, on the difference operation, we see that the green of the cylinders is "pasted" onto the inside of the geometry. So, instead of simply clipping the span for the minuend object, we replace one of the minuend span's hits with whichever hit in the subtrahend span lies at the newly-created boundary. However, the hit cannot be slotted into the new span unchanged. Leaving it untouched leads to artifacts where the normal are flipped. To combat this, whenever we do this kind of "pasting" logic, we negate the normal of the hit from the subtracted geometry, ensuring our normal stays geometrically valid.

4 Results

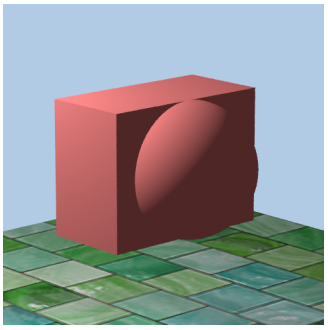

Figure 6: Union operator

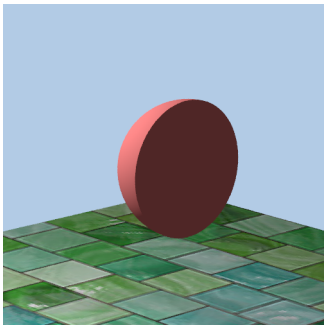

Figure 7: Intersection operator

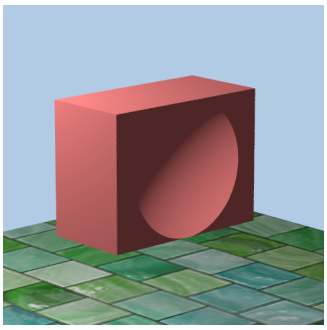

Figure 8: Difference operator

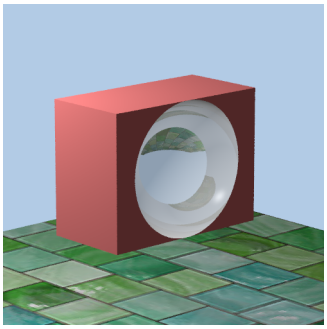

Figure 9: Difference operator with reflection

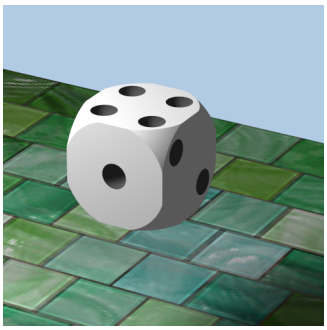

Figure 10: A die, using intersection and difference. Note that the pips are carved rather than decals.

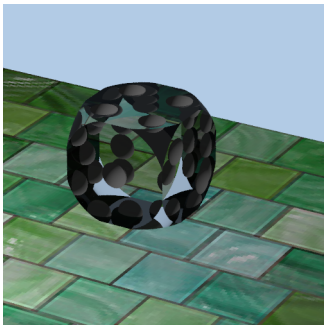

Figure 11: Unfortunately, CSG objects don't work as nicely with refraction as they do with reflection

4.1 Performance

Scene: dice (600x600px)

AA samples	Shadow samples	Render time
0	0	8.12s
0	1	11.46s
0	16	73.08s
16	0	102.5s
16	16	1135.16s

(Render time was measured with GNU `time` using an Intel Core i7-6700HQ CPU)

5 Future Work

While we are satisfied with the results, this is still a fairly simple project with much room for improvement and/or expansion. The following potential improvements have been roughly partitioned into the following three categories:

Interactivity: As it stands, CSG trees must be specified by manually editing an OBJ in a text editor, as the modified format is unrecognizable to other programs. Since there do not appear to be any open standards for the encoding of solid models, the alternative is making objects easier to create in the program itself. Specifically, the addition of gizmos to move elements around, and the ability to anchor children to parents (so moving the root moves the entire tree) would help greatly.

Performance: This program was created mostly as an exploration, and was not implemented with speed as a primary goal. This is especially clear with the nearly 20 minutes taken to render the

Due to the nature of CSG objects as a fairly dense cluster of primitives, the ability to quickly determine that a ray intersects with none of the primitives would save a lot of calculation. The simplest way to do this is with bounding boxes, especially since ray-box intersection is already implemented. (If the ray doesn't intersect with the minimal box which bounds all of its leaf primitives, then it cannot possibly intersect with the object.) This is closely related to two other potential performance enhancements: short-circuiting and a general spatial accelerator.

Expansion: solid models have several interesting properties that were not explored. Among them:

Since every point in the scene can be classified as inside or outside the volume, the epsilon problem can instead be replaced entirely by keeping track of whether a ray is cast into or out of a solid. When a ray is cast into a solid, all spans where $ray\_dir \cdot N_{surface} < 0$ should be discarded, and rays cast into the air should ignore spans where $ray\_dir \cdot N_{surface} > 0$, where $N_{surface}$ is the surface normal at a span's nearest intersection point. When implementing, care should be taken that the $N_{surface}$ is truly the surface normal of the entire CSG object, and not merely the normal of the primitive it intersects with.

Finally, since solid models can be carved and manipulated in ways that make 2D textures difficult to make sense of, they are perfect candidates for procedurally-generated procedural textures as presented in (Perlin 1985).

6 Conclusion

In this project, we were able to extend our existing ray tracing engine to support constructive solid geometry. By expanding out ray casting to record span information about the rays' entire trajectory through the primitives in a given CSG object, we were able to implement the union, intersection, and difference operator at the level of these spans. Then, after casting rays through all the leaf primitives in the constructive solid geometry tree, apply the operations from the bottom of the tree, filtering up to the top. By being careful with managing normal and material information, we were able to crystallize these spans into a single true intersection point for near-seamless integration into the existing ray tracing code.

References

Chen, M. and J. V. Tucker (2000). "Constructive Volume Geometry." In: *Computer Graphics Forum* 19.4, pp. 281–293.

Duff, T. (1992). "Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry." In: *ACM/SIGGRAPH Computer Graphics* 26.2, pp. 131–138.

Foley, J. D. et al. (1990). *Computer Graphics: Principles and Practice.* Addison-Wesley.

Perlin, Ken (July 1985). "An Image Synthesizer." In: *SIGGRAPH Comput. Graph.* 19.3, pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: http://doi.acm.org/10.1145/325165.325247.

Requicha, A. A. G. (1977). *Mathematical Models of Rigid Solids.* Technical Memo 28. Production Automation Project, University of Rochester.

Roth, S. D. (1982). "Ray casting for modeling solids." In: *Computer Graphics and Image Processing* 18.2, pp. 109–144.

Wyvill, B., A. Guy, and E. Galin (n.d.). "Extending the CSG tree: warping, blending, and Boolean operations in an implicit surface modeling system." In: *Computer Graphics Forum* 18.2 (), pp. 149–158.

Zhou, Qingnan et al. (2016). "Mesh Arrangements for Solid Geometry." In: *ACM Transactions on Graphics (TOG)* 35.4.