Algorithm for Binary Operations on Solid Geometry Represented by Halfspaces

Halfspace-rep is a scheme in which solid geometry is represented in terms of a region of space enclosed by planes (i.e. the common intersection of a collection of halfspaces). Such a solid must be convex.

The algorithm given here is believed to be conclusive as it has been thoroughly tested in a variety of circumstances. It does not always result in an optimal solution, either in terms of the number of faces, or the number of brushes in the result.

To avoid confusion, a region of space enclosed by a number of planes will be termed a *brush* and a collection of brushes will be termed an *object*. Since the result of a solid set operation on two brushes is not necessarily convex, it must be given as an object which consists of zero or more brushes.

Listing 1 describes the algorithm. It relies on the fact that when a brush is divided by a plane, both segments are themselves brushes, since they are still regions bounded by planes.

Given two input brushes `A` and `B`, the `difference` (`A-B`), `intersection`, or `union` can be returned. The algorithm recursively divides brush `A` by the planes of brush `B`. After each division, the front region is added to the `difference` object, and the back region is considered to be brush `A`.

Note that a polygon should only be considered in front of a plane if all of its vertices are in front of that plane. Note also, that `αpolygon` is the part of `α` behind all other planes in `A`.

```
let difference = new object
let intersection = new object
let union = new object

for β = each plane in brush B
{
   let frontbr = new brush
   let backbr = new brush

   for α = each plane in brush A
   {
      let αpolygon = α clipped against all other planes in A

      if αpolygon is in front of β: add α to frontbr
      if αpolygon is behind β: add α to backbr
      if αpolygon is divided by β: add α to frontbr and backbr
   }

   if backbr ≠ ∅ and frontbr ≠ ∅:
   {
      add β to backbr
      add (β with inverse normal) to frontbr
   }

   set A = backbr
   add frontbr to difference
}
add A to intersection
union = difference
add B to union
```

Listing 1

Figure 1 shows an example, viewed orthographically. `A` is shown in blue and turquoise, and `B` in green and turquoise in (a). In (b), the situation during the execution of the first `for` loop is shown. The orange segment shows `frontbr`, which is added to `difference`. (c) shows the situation several cycles later. In (d), the final `difference` is shown is orange, and the `intersection` in turquoise. Note that the brushes in `difference` are never divided by β.

The algorithm returns a correct result for any arrangement of `A` and `B`, including when `A`⊃`B`, when `A`⊂`B`, when `A` and `B` partially intersect, and when `A` and `B` do not intersect at all.

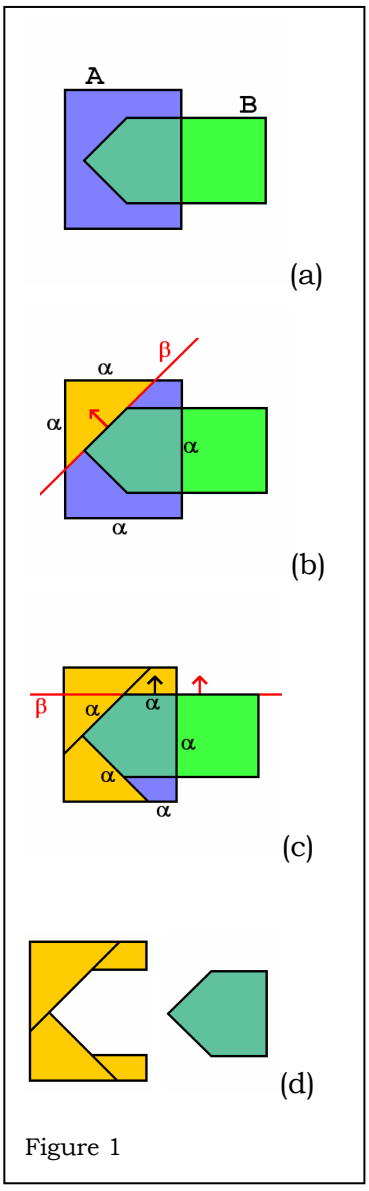

Figure 1

In Figure 1(c), β and an instance of α are coplanar. There are two ways in which β can be coplanar with α - their normals can have identical or opposing directions. In both situations, β should be considered in front of α. However, a further optimisation can be applied to the algorithm; when α and β are co-planar with opposing directions, `A` does not intersect `B`, and the result is known.

Figure 2 shows rendered images of a difference operation, performed using this algorithm.

In order to calculate the result of a solid set operation on two objects, every brush in one object must be compared with every brush in the other object using the algorithm in Listing 1. Simple bounding sphere tests may be used to avoid unnecessary comparisons.

Images of Implemented Software

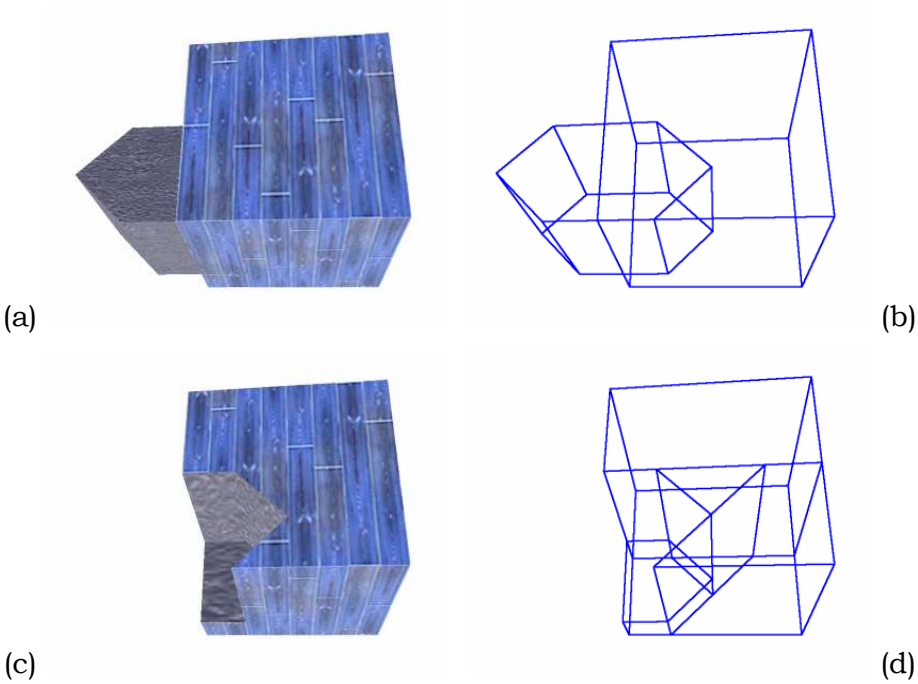

Figure 2 - a solid set operation, performed using the algorithm described above.

(a) and (b) show the initial position of two brushes, rendered as textured solids and wireframes respectively. (c) and (d) show the resultant brushes after a difference operation.