Constructive Solid Geometry Theory and Implementations

Duo Tao

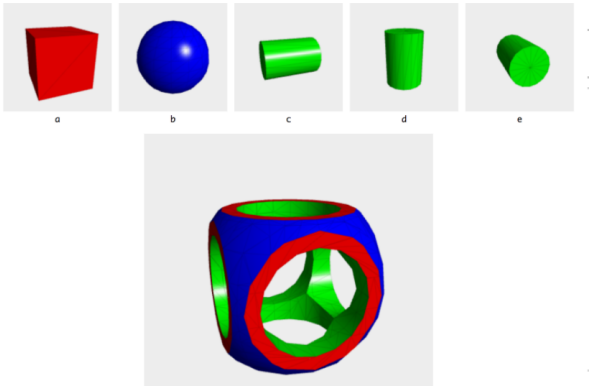
Professor Josh Davis
CS311: Computer Graphics

March 15, 2017

1 Constructive Solid Geometry

Constructive Solid Geometry is a way to create more complicated meshes from simple meshes. The simple meshes are called primitives. They can be spheres, boxes, shells, capsules etc. Basically, they are easy to define as a concept and their shape are generally easy to imagine. As for implementations, we usually hardcode their dimensions and shapes in our implementations.

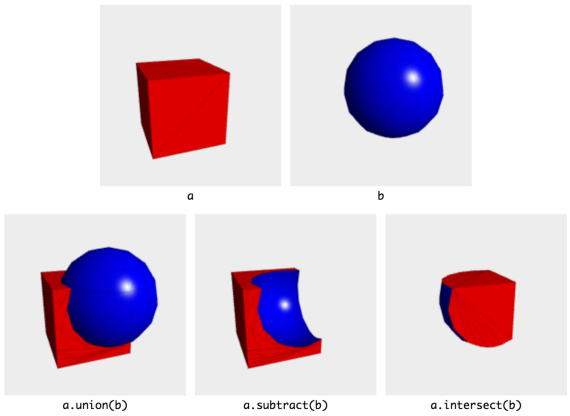
An example of CSG is given below:



In this example, we can see there are five primitives, one cube, one sphere and three cylinders oriented in different directions. Below them is a mesh that can be constructed from those five primitives. We can see that the resulting mesh is very complicated to define piece by piece, but we can build it easily from those simple primitives with CSG operations, which is covered in the next section.

2 CSG Operations

There are three types of CSG operations: Union, Intersection and Subtraction. The concept of those operations is just like the intuitive way of getting the merge, intersection or difference between two solid bodies. Examples of the three operations are shown below:



There is a cube in (a) and a sphere in (b). When we union them, we get their union, which is in the left picture of the second row. When we subtract b from a, we get the second. When we do intersection between them, we get the third. It is just the intuitive way of how solid bodies should work.

An interesting aspect of this is that the intuitive way to look at the meshes merge/intersect/subtract becomes unintuitive when we deal with meshes. In other words, when we are implementing the operations, we are not dealing with solid bodies. We are dealing with meshes of triangles. There is nothing but a surface there. Therefore, we need to think about some trick to simulate the effect of a solid body merge with only surfaces. We need to get the surface so as to simulate the effect. That is an interesting part of the implementations.

3 My Project

My project is all based on an open source javascript library called csg.js. (Its Github link is cited at the end)[1]. My project has two parts:

- Theoretical Part: I need to fully understand the theories of CSG by doing a close reading of the code of csg.js to understand how CSG works. I will derive a high level workflow of CSG from the details of implementations in the code, and this part results in a project report, which is this paper.
- Coding Part: Another part is the coding part. For the coding part, I want to implement CSG in C with OpenGL 3.2. In other words, I will rewrite the code and thus port the javascript library csg.js into C. We will also mention a few things about this part of work in the result.

4 How CSG works on a high level

The inputs of CSG are meshes, which are just a list of 2D shapes (triangles in our case, polygons in csg.js). CSG does boolean operations on the meshes and output another mesh. Since all meshes, no matter a primitive or a mesh built from another boolean operation, should be essentially the same thing, they should both serve very well as the inputs of a CSG boolean operation. Therefore, the essence of CSG is the implementation of the boolean operations between meshes. The boolean operations should be general enough for any mesh, not just primitives.

Now that our problem becomes "how does the boolean operations work", we can examine the three types of operations. For example, the input of union is two meshes, and it outputs another mesh representing the union of the inputs. It turns out that all the three boolean operations can be implemented in a similar way. On a high level, the procedures are like this when the inputs are a set of meshes and a boolean formula that represents the CSG operations between them:

1. Build the BSP (Binary Space Partitioning) trees for each mesh.
2. Do the operations on the BSP trees.

3. Output the polygons from BSP trees.

The first step is the most theoretically important. We will elaborate on BSP trees in the next section. Once we have the BSP trees, we will be able to do 2 and 3 relatively easily because we can make use of the operations of the BSP tree.

5 BSP Tree

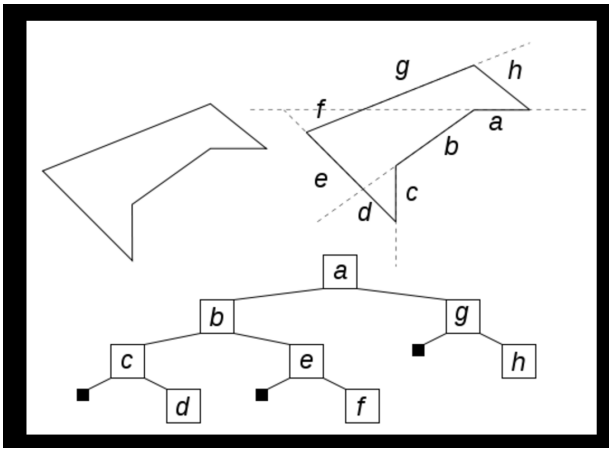
5.1 Definition and Structures

BSP tree is Binary space partitioning tree. It is a binary tree to organize polygons. Conceptually, each node represents a plane that split part of the space into half. In terms of implementations, there are two items in each node of the tree: a plane and a list of polygons or triangles on that plane. The plane splits the node in half and the left subtree represents the space in front of the plane and the right subtree represents the back. Front and back are determined according to the normal vectors. Therefore, the root of left subtree splits the space in front in half.

A node becomes a leaf when there is no polygons or triangles in front or behind the plane. Or other words, All triangles in this part of the space are coplanar. In most cases, this would mean that there is only polygon in this space.

We do not have to worry about the case when a polygon cross the plane because, in that case, we just split the polygon into two and one part stays on the front side, the other part on the backside.

To illustrate the structure of a BSP tree, we give an example BSP tree structure from Wikipedia is shown as below.[2]



This example gives a dimensionally squeezed BSP tree usage. In our case, we use the tree for 3D meshes so the polygons will be in 2D. In this case, we can think of everything going down by 1 dimension so the meshes are 2D shapes and the "polygons" for the meshes are thus 1D, which are the edges of the 2D shape. A plane splits a 3D space into half and a line splits a 2D plane into half. Therefore, there is no obstacle to understand the usage of BSP in 3D with this picture since we just go up by one dimension.

In this example, line a splits the plane in half. One side has g, h in it and the other side has b,c,d,e,f in it. We pick b as the root of the second half and thus it has c,d on one side and e,f on the other. When we go down to a leaf like d, we find that it is the only edge in its space so it becomes a leaf in the tree. Also, we might want to note that maybe in the original input, e and d are combined as one line, it is split into two during the construction of the BSP, which will be elaborated in section 6.

5.2 Operations

We know that a BSP tree represents the organization of a list of polygons with binary partitions of the space. Based on this, we can have the following operations on BSP tree, based on what I learned from csg.js.

1. clipTo(AnotherMesh): remove the part of this mesh inside another mesh.

2. invert: invert the solid / empty space represented by this mesh.

3. allPolygons: output all polygons as a list

We use the three operations combined to conduct a boolean operation on meshes. The third one is just an output method. Once we constructed the BSP that represents the result of the boolean operations, we can output the result as a mesh, which is a list of polygons, instead of a tree. Some of the methods are implemented in interesting ways, which we will cover in section 8.

5.3 To understand BSP Tree as a Data Structure

Based on the facts we know about BSP tree, we give two pieces of intuitive ways to understand BSP tree as a data structure.

First, a BSP tree is a way to organize the polygons. It does not change the mesh; it only organizes the mesh in a better way. It does not mean that we do not change in terms of data but that we does change conceptually. For example, we split a triangle into two. That changes the underlying data: we get one more triangle. But it does not change what the mesh represents. We are given a list of polygons and we want to organize them in a way that is good for CSG operations. Because CSG operations require a lot of testing inside/outside, a fine binary split of the space helps a lot. For example, if we want to know if a point is inside a cube, we can build the BSP tree that splits the space into 16 pieces. Being inside the tree means being in one of the pieces and thus go into a child of a leaf. This approach helps the testing of inside/outside and thus enable the tree to support operations like clipTo as is mentioned above. So organizing the polygons in this way will thus be good for CSG operations.

Second, a BSP tree is also a way to split the space. Since each node represents a plane and each tree structure represents a split of the space into half with that polygon, a whole represents a split of the space. This intuition helps thinking about operations on the tree. We can think of adding a polygon into the tree as "melting" the polygon with a metal mould as it fits into the split of the space and adds new splits into the original mould.

6 Build a BSP Tree from a list of Polygons

If we are given a mesh, which is a list of polygons. We do the following to build a BSP tree that represents the mesh, as is learned from csg,js[1]:

1. If the input list is empty, we return null.

2. Pick a polygon as the root. Partition the space based on the plane of that polygon. All other polygons fall into one of the four categories:

    - Front: when the polygon is totally in front of the plane. We put it into a list which goes into the recursion that builds the left subtree.
    - Back: Similar as front.
    - Coplanar: By the structure of a BSP tree, we know we should add into this node.
    - Cross: We clip it into two polygons. One goes into the front recursion list; the other goes into the back recursion list.

By the characteristics of a BSP tree as we discussed in the last section, we know that this results in a valid construction of the tree. Note again that the BSP does not change the mesh. It does not add or remove any parts from the mesh. It just reorganizes it, so when we draw all the polygons in the tree as a new mesh, the new mesh is the same as the original mesh.

7 Using BSP Trees for CSG

Now that we know how to build a BSP for a mesh, we proceed on discussing how to use the BSP trees constructed for each mesh to do the boolean operation.

1. **Union(A,B) = B.clipTo(A) + A.clipTo(B).** We just add up all polygons we get from the meshes clipping each other. This is reasonable because the parts of the mesh inside another mesh are never seen. Moreover, we need to remove them because, if we do not, their existence prevents the mesh from being a valid input for the next boolean operation. They also violate the concept of merging the meshes as if they are solid bodies. So we have to remove them, even though we get the same visual effect if we only look at the union results.
2. **Intersection: invert(union(invert(A), invert(B)))**
3. **Subtraction: invert(union(invert(A), B))**

We can understand subtraction and intersection based on the same approach we have for union: representing exactly the surface of the resulting mesh when the two meshes interacts as solid bodies.

8 Some Interesting Implementation Details

When I am reading the implementation details of csg.js, there are some implementation tricks I found very interesting. Even though we do not intend to get to every implementation details in this paper, it will be helpful to know about some of those to eradicate lingering confusions about the concepts.

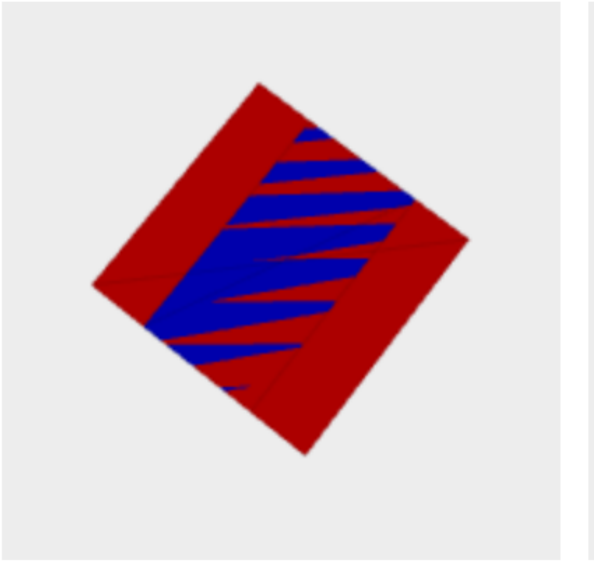
1. **Triangles vs. Polygons.** The two are equivalent, with triangle being a more low-level thing and the polygon a more general and top-level thing. I am using triangle meshes, just like the ones we use in class. The javascript library csg.js uses polygon meshes, the smallest unit in the mesh is not necessarily a triangle but a polygon, which can be a square or something. We know that any polygon can be split into triangles so the power of both meshes are the same. Actually, in the javascript library, before it passes the polygons into WebGL (the equivalence of OpenGL for webapps), it also converts the polygons into triangles.
2. **Testing insideness in clipTo()**. If we have a convex mesh, it is easy. The BSP for the mesh is a right-going linked list. If polygon is on the backside of every plane (or rather, on the right side of the rightmost child), it is inside the mesh. However, what if we have a concave mesh. We have the ability to make sure all primitives a concave, but there is no way to ensure that the users only build concave meshes from the boolean operations. Since the boolean operation result of two valid meshes is a valid meshes as an input, we need to support concave meshes.

   After reading the code closely, I realized something profound: **a concave mesh can be split into some convex meshes.** Since we are splitting the mesh with all planes involved in the mesh, we must eventually split the concave mesh into some convex meshes. We know how to test insideness of a polygon for a convex mesh so we refine the testing of insideness using a BSP tree like this: being the right most child of a subtree. It means there exists some parts of the mesh after being split that contains the polygon. Then the polygon is inside the mesh.
3. **Implementation of invert()**. We know from above that inverting node means switch the solid and empty space that the mesh represents. The trick to implement this flip all planes and triangles in the node. Also, we need to swap all of the left / right subtrees. This becomes clearer after we think carefully about the relation between the mesh and the tree structure. For example, we need to swap the left and the right subtree because we flipped the plane. Since front and back are defined with the direction of the normal vector of the plane, we know that what was originally in front is in back now.
4. **Numerical noise for testing coplanar.** This is a very important consideration since we get new triangles and new vertices via interpolation, there must be numerical noise caused by the imprecision of computers. Polygons that are supposed to be on the same plane might differ by a very small value. If csg.js and my implementation C, we use a gap difference of 10^{-6} from the space as being on the space.

5. **More about Union Implementation.** If you look at my code or the javascript library, you might notice that we are doing more than what is described in section 7 to implement union. That is because we need to consider coplanar polygons. This is best explained in the comment of the library[1]:

> The only tricky part is handling overlapping coplanar polygons in both trees. The code above keeps both copies, but we need to keep them in one tree and remove them in the other tree. To remove them from ‘b‘ we can clip the inverse of ‘b‘ against ‘a‘.

I tested. If we do not do this, we show both copies that might have different colors and end up getting something like this instead:



We can see both copies, the red from mesh a and the blue from mesh b, are shown.

9 Conclusion

In this project, we learned about the concept of CSG, elaborated on how we do it for triangle meshes via BSP trees and talked in details about BSP trees. Basically, GSC is a way to build complex meshes from primitives. It takes in some meshes, builds BSP trees for each, do boolean operations based on the trees and dump the resulting tree to get the new mesh.

If we look further into this project, I think we can think of the case when we have a very long boolean operations so we might want to first simplify the boolean operations to minimize the computational cost before we execute. That could lead to better efficiency and support more automated and high-level CSG programs.

References

[1] Evan Wallace. *CSG.js*. URL: https://github.com/evanw/csg.js/.

[2] Public Collaborative work. *Binary Space Partitioning*. URL: https://pl.wikipedia.org/wiki/Binary_space_partitioning.