24-681: Computer-Aided Design
Spring 2013

Constructive Solid Geometry Using BSP Tree

Christian Segura[1], Taylor Stine[2], Jackie Yang[3]

Abstract
Constructive solid geometry (CSG) is a pivotal component of CAD and CAE packages. CSG allows us to represent complex shapes and models as a series of Boolean operations between primitives. For example, punching a hole through a cube would be difficult to represent with an implict or explicit funciton. The CSG algorithm we have developed allows something like this to be represented as a simple Boolean operation between a cube and cyllinder. Here we present an implementation of CSG using an efficient spatial datastructure called a binary space partitioning tree (BSP tree). These BSP trees allow us to perform Boolean operations on complex models in a matter of milliseconds. In this paper, we validate our concept and implementation by performing Boolean operations on a series of intracate models. The results show our algorithm is efficient and accurate.

Keywords
Computer-aided design(CAD) — Constructive solid geometry(CSG) — Binary space partitioning(BSP)

[1] *csegurar@andrew.cmu.edu*
[2] *tstine@andrew.cmu.edu*
[3] *jackiey@andrew.cmu.edu*
Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, United States

Contents

1. Problem Summery

Constructive Solid Geometry is a solid modeling technique that allows a user to create a complex surfaces or volumes by using Boolean operators. In doing so the user can combine different geometry as they seem fit to generate a result.

Multiple techniques exist in CAD packages and graphics applications which allow the creation of geometric models. CAD packages and computer graphics applications tackle different geometry problems with different approaches. For instance, voxel modeling can be used to represent three dimensional information, while boundary representation can be used for mesh generation. However, the best technique to create geometries from existing geometries is CSG.

CSG offer many advantages over other computational geometry methods. The main one is that it allows the user to perform basic operations on simple models that result in complex yet accurate geometric objects. For instance creating a single layer of a gyrocube, would be prove to be quite complex with other methods. With CSG, we can easily find the intersection of a cube and sphere, the union of three differently oriented cylinders, and then finally subtract the cylinders' composite model from the first one to build a 3D single layer of a gyrocube. Figure 1 shows the process described above. CSG also allows efficient detection of various geometric characteristics within 3D models, such collision detection and water tightness. Collision Detection (Figure 2) can very often be a computationally expensive process. Using CSG reduces the collision detection computational cost by instead spending those resources up front in creating a more efficient data structure which yields faster results when testing for intersections/collisions between models. Furthermore, some model characteristics can be inherited from base geometries after Boolean operations. One can easily check if a model is water tight by checking if the components used to create it are also water-tight. This prove to be very helpful when characterizing certain geometric information which is not immediately obvious or is computationally expensive to calculate on a newly created model. Another important use for CSG is efficiently determining the visibility of models relative to each other with a changing viewpoint. This offers graphics processors a powerful and efficient method to render front objects without the occlusion of the back ones. The binary space partitioning used for CSG Boolean operations facilitates this function (Figure 3).

CSG can be used for multiple applications, but must remain watchful of its pitfalls. CSG can be implemented in various manners, but the programmers must ensure that computational costs don't exceed the processing power or allocated loading time. The smarter method encourages data restructuring inside a BSP tree, which ensures the computational cost is paid for up front instead of doing so when

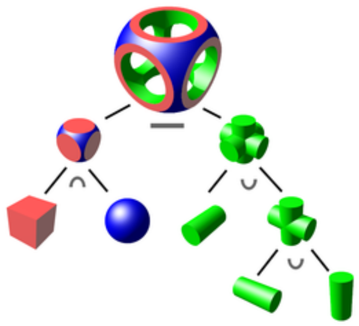


Figure 1. Example of CSG tree [6]

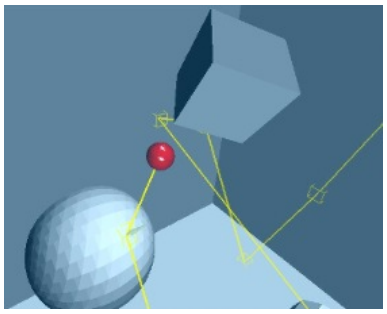


Figure 2. Collision detection [7]

performing each of the different Boolean operations.

2. Algorithm

One of the key problems in computer aided design and graphics is determining what objects are visible relative to one another, with a constantly changing viewpoint (Figure 4). Furthermore, the problem of surface to surface interference is difficult to classify for models with a complex geometry. When a scene or rendering consists of several models, this problem becomes even more difficult.

In order to resolve these issues, graphics software engineers utilize a spatial data structure known as a binary space partitioning tree (a.k.a. BSP tree). The BSP tree represents a way to recursively divide a scene along two sides of a plane, until some partitioning criterion is met. There are many similar data structures known to computer graphics, (octrees, k-d tree, bounding box hierarchy etc.) but BSP trees provide us with the most flexibility in partitioning our scene. That flexibility results from our ability to orient the partitioning plane in any direction, without being constrained by orthogonality as in octrees or k-d trees.

2.1 Create BSP tree

The construction of a BSP tree is simple to visualize. Image a scene consisting of three triangles. The key properties of the implicit planes these triangles define in 3D space is that for all points on one side of the plane p^+ we can easily create a function $f_1(p^+) > 0$. Similarly for all points on the other

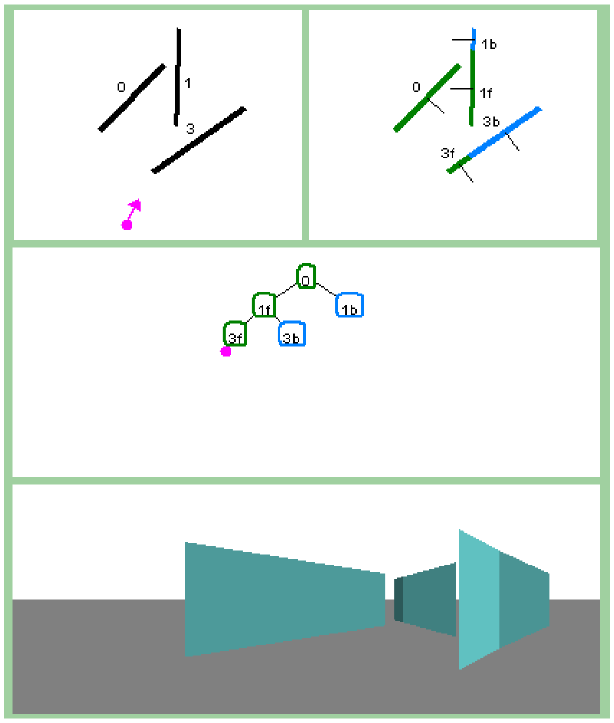


Figure 3. Render objects using BSP tree

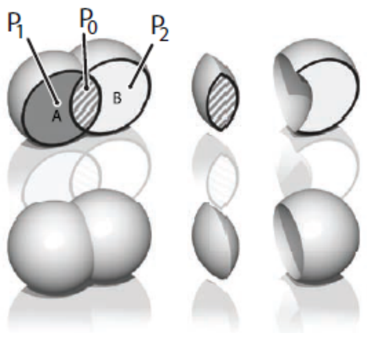


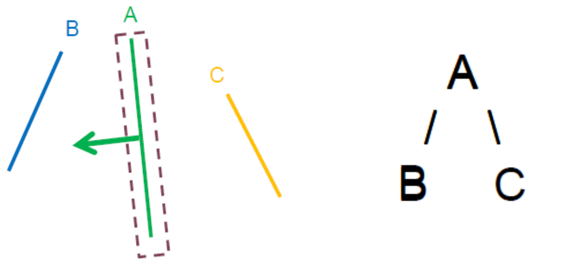
Figure 4. Concept of CSG in 3D

side of the plane, $f_1(p^-) < 0$. Using this property of implicit planes we can define on which side of the plane a triangle (consisting of three points) lies. Initially, let us assume that all of the triangles in our scene are either on one side of our partitioning plane defined by our triangle (Figure 5, Figure 6). We can pick one of the triangles and partition the other triangles about it with pseudo code in Algorithm 1.

This example can be generalized to many object, again assuming the simple case where none of our polygons span our dividing plane. In this example, $f_1(p)$ is the implicit function of a plane created by triangles with counter clockwise vertices a, b, and c:

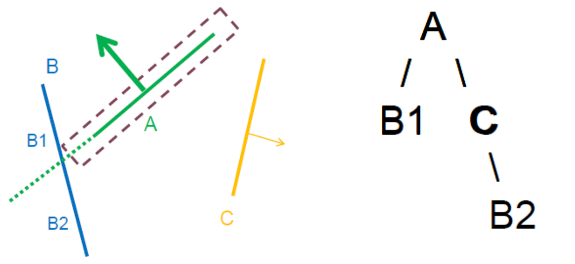
$$f_1(p) = ((b-a)\times(c-a))\cdot(p-a) = 0 \tag{1}$$

However it can be faster to store the values of the implicit



(a) Multi-plane in 3D space (b) BSP tree representation

Figure 5. BSP tree creation (without intersection)



(a) Multi-plane in 3D space (b) BSP tree representation

Figure 6. BSP tree creation (with intersection)

Algorithm 1 Triangle partition pseudo code

```
Require: $partition = triangles[0]$
for $triangle = triangles[begin \rightarrow end]$ do
    for $p = points[in \triangle]$ do
        if $f_1 < 0$ then
            return in back of plane
        else if $f_1 > 0$ then
            return in front of plane
        end if
    end for
end for
```

plane equation in the form:

$$Ax + By + Cz + D = 0 \tag{2}$$

This is the same expression, and can be faster to solve than equation (1). Here the constant D is equal to:

$$D = -n \cdot a \tag{3}$$

Storing the equation in this form can reduce some computation time associated with taking the cross product. Thus, this naturally leads to the follow pseudo-code for BSP tree construction shown in Algorithm 2.

Algorithm 2 BSP tree initial pseudo code

```
Require: $tree.node = triangles[0]$
for $i = 2 \rightarrow N$ do
    tree.add(triangles[i])
end for
function ADD(triangle T)
    if $f(a) < 0 \wedge f(b) < 0 \wedge f(c) < 0$ then
        if back-subtree does not exist then
            create back-subtree
            back-subtree.node = T
        else
            front-subtree.add(T)
        end if
    else if $f(a) > 0 \wedge f(b) > 0 \wedge f(c) > 0$ then
        if front-subtree does not exist then
            create front-subtree
            front-subtree.node = T
        else
            front-subtree.add(T)
        end if
    end if
end function
```

From our above constraints on building the tree (i.e. none of the triangles span the tree) we have not yet defined how to handle the case of when some functions of the vertices of the triangle are positive, and others are negative. In this case, the only thing we can do is split the triangle into three new triangles (Figure 7).

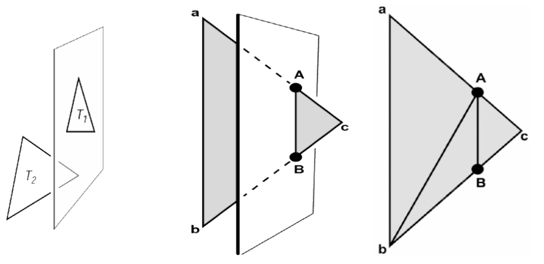


Figure 7. Principle to split a triangle

Assuming that a and b are always on one side of the triangle and c is on the other the new triangles will always be equal to:

$$\begin{aligned} T_1 &= (a,b,A) \\ T_2 &= (b,B,A) \\ T_3 &= (A,B,c) \end{aligned} \tag{4}$$

It is clear from this representation that a special case will emerge when the triangle is split perfectly down the middle. Although rare, We will account for this by not splitting the triangles if this occurs. Thus our final implementation of the BSP tree creation is shown in Algorithm 3.

The last component of building our BSP tree is computing A and B. Computing the A and B intersection points consist of simply solving a ray-plane intersection equation:

$$\begin{aligned} p(t) &= a+t(c-a) \\ n\cdot(a+t(c-a))+D &= 0 \\ t &= -\frac{n\cdot a+D}{n\cdot(c-a)} \\ A &= a+t(c-a) \end{aligned} \tag{5}$$

Our creation algorithm for a BSP tree is now complete.

2.2 Merge two trees

Because of their ability to divide a series of polygons by an arbitrarily chosen plane, BSP trees offer the ideal data structure to perform Boolean operations on arbitrary pieces of geometry. In order to perform these operations, we have developed an algorithm that ”merges” two BSP trees. Assuming a simple case of just two models in a scene with which we want to perform Boolean operations, the first thing we need to do is create BSP trees for both of the geometries. These trees are created independently for each geometry initally, so we will only be testing polygons in one model for each BSP tree creation. When we create these BSP trees, instead of simply arbitrarily choosing planes to divide the polygons by, we will choose to divide the polygons by the polygons on the surface of the model. Thus these BSP trees allow us to create an efficient structure to traverse the boundaries of a complicated mesh. Now that we have two representations of the boundaries of the models, we begin to merge them by traversing one of the trees to obtain a list of polygons that represent the surface boundary of one of the models. We use this list of polygons because it contains new polygons created by splitting the polygons of the model by the dividing planes chosen when creating the tree. These could not be captured if we just used the list of polygons provided with the model for the next step in the merging algorithm. The traversal of the BSP tree is quite simple as shown in Algorithm 4.

This will give us a list of polygons represented as a preorder traversal of the tree. Now that we have a list of polygons from the tree that represents model A, we have to push these polygons through the tree of model B so that we can see how they will be classified according to B’s dividing polygons. Algorithm 5 looks very similar to the add function listed above. This algorithm assumes that we are working with models made up of only triangles.

Algorithm 3 BSP tree final pseudo code

```
Require: $fa = f(a)$, $fb = f(b)$, $fc = f(c)$
 1: function ADD(triangle T)
 2:     if $|fa| < \varepsilon$ then
 3:         $fa = 0$
 4:     end if
 5:     if $|fb| < \varepsilon$ then
 6:         $fb = 0$
 7:     end if
 8:     if $|fa| < \varepsilon$ then
 9:         $fb = 0$
10:     end if
11:     if $fa \leq 0 \wedge fb \leq 0 \wedge fc \leq 0$ then
12:         if back-subtree does not exist then
13:             created back-subtree
14:             back-subtree.node = T
15:         else
16:             back-subtree.add(T)
17:         end if
18:     else if $fa \geq 0 \wedge fb \geq 0 \wedge fc \geq 0$ then
19:         if front-subtree does not exist then
20:             create front-subtree
21:             front-subtree.node = T
22:         else
23:             front-subtree.add(T)
24:         end if
25:     else
26:         cut the triangle
27:     end if
28:     compute A
29:     compute B
30:     $T_1 = (a,b,A)$
31:     $T_2 = (b,B,A)$
32:     $T_3 = (A,B,c)$
33:     if $fc \geq 0$ then
34:         back-subtree.add($T_1$)
35:         back-subtree.add($T_2$)
36:         front-subtree.add($T_3$)
37:     else
38:         front-subtree.add($T_1$)
39:         front-subtree.add($T_2$)
40:         back-subtree.add($T_3$)
41:     end if
42: end function
```

Algorithm 4 Traversal of the BSP tree

```
function TRAVERSE(tree)
    if tree does not exist then
        exit
    end if
    polygon-list.add(tree.polygon)
    traverse(tree.back-subtree)
    traverse(tree.front-subtree)
    return polygon-list
end function
```

In Algorithm 5, instead of adding these triangles to the tree, we are traversing the tree to see where they belong in the BSP representation of the model. If we get to a point where the triangle is classified as being behind the current dividing plane and there are no there are no other dividing planes in the back-subtree of the BSP tree, then we can classify this triangle as being inside of the model. The same can be said about a triangle completely in front of the dividing plane. An interesting case emerges when we find that the dividing plane we are testing splits the triangle. In this case, we cannot yet classify the newly split triangles as completely inside or outside of the polygon, because we could have split a triangle that may intersect the dividing plane, but isn't split by the boundary of the model. The only way to classify these newly created triangles is to push them through the tree again. Once all of the triangles have been inserted, we now have a list of triangles from model B that are inside/outside of model A. We perform this same procedure for model A, to obtain two similar lists. Once we have these lists, it is a trivial matter to combine them to obtain the Boolean operations we desire (Figure 8).

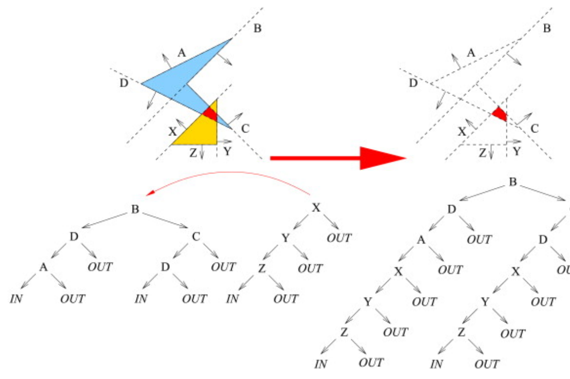


Figure 8. Merge two trees [8]

The reason we choose to use BSP trees and use Algorithm 5 is because of the efficiency of merging the two trees. The operation of merging two trees takes milliseconds, and can be done in O($n \log n$) efficiency in the best case and O(n^2) in the worst case where we have a very unbalanced tree. The most computationally expensive portion of the Algorithm 5 comes from actually creating the trees. For convex polygons

Algorithm 5 Push polygons from tree to model

```
for $i = 2 \to N$ do
    insert($tree_B.root$, $A_{triangles}[i]$)
end for
function INSERT(Node node, triangle T)
    $fa = f(a)$
    $fb = f(b)$
    $fc = f(c)$
    if $|fa| < \varepsilon$ then
        $fa = 0$
    end if
    if $|fb| < \varepsilon$ then
        $fb = 0$
    end if
    if $|fc| < \varepsilon$ then
        $fc = 0$
    end if
    if $fa \leq 0 \wedge fb \leq 0 \wedge fc \leq 0$ then
        if back-subtree does not exist then
            inside-model.add(T)
        else
            insert(node.back-subtree, T)
        end if
    else if $fa \geq 0 \wedge fb \geq 0 \wedge fc \geq 0$ then
        if front-subtree does not exist then
            outside-model.add(T)
        else
            insert(node.front-subtree, T)
        end if
    else
        cut the triangle
    end if
    compute A
    compute B
    $T_1 = (a, b, A)$
    $T_2 = (b, B, A)$
    $T_3 = (A, B, c)$
    if $fc \geq 0$ then
        insert(node.back-subtree, $T_1$)
        insert(node.back-subtree, $T_2$)
        insert(node.front-subtree, $T_3$)
    else
        insert(node.front-subtree, $T_1$)
        insert(node.front-subtree, $T_2$)
        insert(node.back-subtree, $T_3$)
    end if
end function
```

Algorithm 5 can take up to O(n^2) time, because this creates a very unbalanced tree. For models with concavity, the creation of Algorithm 3 can get down to O($n \log n$) efficiency, but can take longer to allocate memory for the new triangles created by the partitioning plane. This is much faster than the O(n^3) case of comparing all planes to all other planes of a naive implementation of model Boolean operations. The pre-processing time is the whole purpose of BSP trees. They are used to pre-cache the information we need in the pre-processing step, to allow for quick traversal when we need to render, merge, or test for collisions. This is the reason they are most often used in video games and CAD systems. Their traversal efficiency allows us to create the Boolean operations for these models in a matter of seconds.

3. Results & Discussion

We downloaded multiple 3D models from the web[1] and put this algorithm to the test in the C++ programming language. They ranged from highly simple 3D shapes like cubes and toruses (Figure 9), to fully complex video game characters models (Figure 10, Figure 11). For each model, we created a BSP tree structure as described in the algorithm above, and then merged each model's tree according to the Boolean operation performed on the pair of models. We tested all the operations, including union, intersection, differences and xor on our models. We used OpenGL to render each model separately on screen first, and then rendered each of the Boolean operation results. Notice that our meshes only consist of surface polygons and contain no volumetric information. As such, you can see into or through our model result who operations cut through the base models. Nonetheless, the results were quite promising.

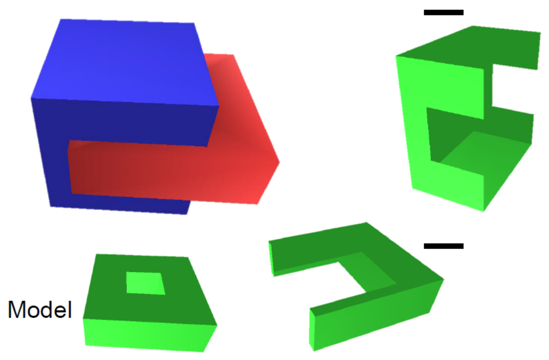


Figure 9. CSG of cube & torus

For many of the simple shapes, our results were very accurate. For instance, CSG Boolean operations between cubes and toruses resulted in error-free models. However, for more complicated shapes, such as the different character models,

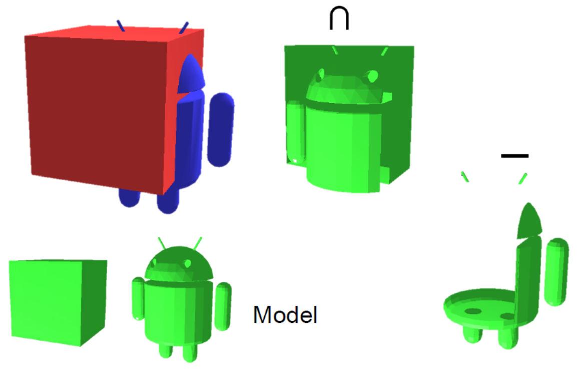


Figure 10. CSG of cube & Android

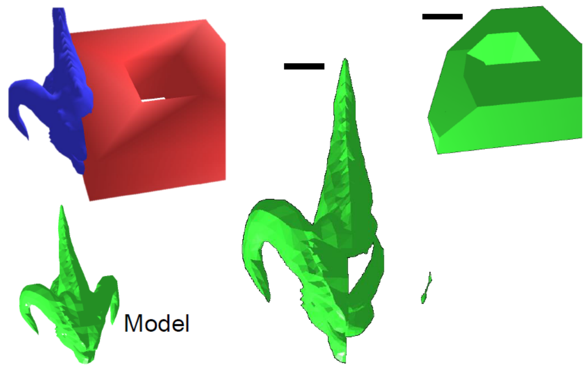


Figure 11. CSG of torus & dragon

the Boolean operation would yield results with minor geometry artifacts. For instance, when creating the intersection between the cube and the werewolf, we get a result which is not quite as expected (as displayed in Figure 12). The artifacts that appear in some of our results appear to manifest themselves when operations are performed on objects with low polygon count. We believe that this occurs due to the lower resolution of one model relative to the other, and thus when the BSP trees of both models are merged, we occasionally are left with extraneous triangles that must get pushed through the BSP tree more than once but end up getting catalogued incorrectly. Nonetheless, the operation combinations we performed impressive results which looked almost identical to results in high computer graphics and computer-aided design applications.

4. Future Work

This project has potential for future work and progress. Due to the brief time period given to the team to work on this project, we limited our code to perform operations on a single pairs of models. Furthermore, we only utilized surface mesh information as inputs and outputs to our algorithm to facilitate its implementation. In the future, we could potentially be able to perform any number of operations on a larger set

[1]`http://www.turbosquid.com`

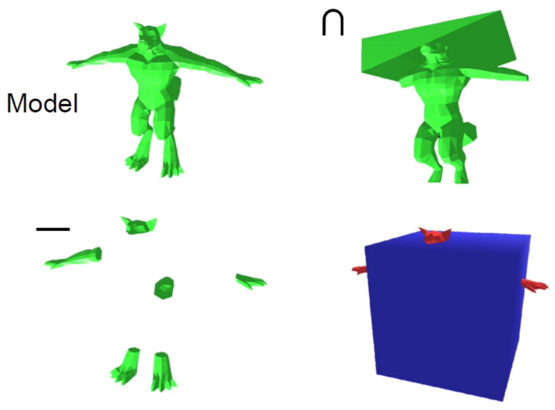


Figure 12. CSG of werewolf & cube

of models consisting of more than two geometries. In doing so, we would enable Boolean operation chaining, where we would created composite models from our base models, and then further perform other Boolean operations on our composite models for more impressive and complex results. In the next stage of our software, we would like to take use voxel representations of our models as input and output to our CSG algorithm. In doing so, we allow OpenGL to also render the volumetric information after performing the Boolean operations on the pair of models. This would enable a higher level of accuracy and an easier visual understanding of our results since we would no longer be rendering just the surfaces of the models, but the a solid in itself (Figure 13). Our algorithm, though efficient, lacks any optimization that can be achieved with tree balancing or parallel programming. Implementing these would further speed up our algorithm execution. Lastly, we would look for different methods to eliminate the geometry artifacts that appear in some of the results. We could automatically increase the number of polygons in a model by splitting surface faces into smaller polygons and get rid of these artifacts entirely. Furthermore, we could attempt to run the algorithm on the polygons themselves instead of having to split the all non-triangular polygons into triangles. This would also speed up the process and help eliminate the root of the artifacts problem.

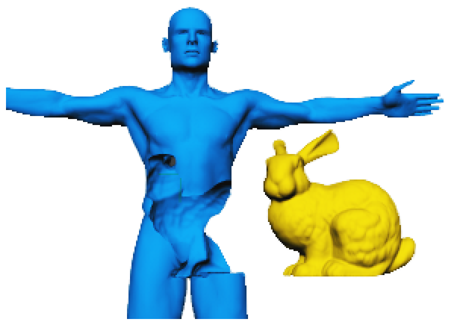


Figure 13. Render solid model instead of surfaces [9]

Acknowledgments

Thanks to Professor Kenji Shimada for his excellent lectures which gave us enough background knowledge to start and finish this project. We appreciate his advice during the semester.

References

[1] B. Naylor, J. Amanatides and W. Thibault, Merging BSP Trees Yields Polyhedral Set Operations, in *Proc. Siggraph '90, Computer Graphics 24(4)*, pp 115-124, 1990.

[2] Miklo Lysenko, Roshan D'Souza and Ching-Kuang Shene, in *Improved Binary Space Partition Merging, CAD, Vol. 40, No. 12*, pp. 1113-1120, 2009.

[3] Shirley, Peter et. al, in *Fundamentals of Computer Graphics, 3rd ed. Wellesley: A K Peters*, 2009.

[4] Tom Duff, Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry, in*SIGGRAPH '92*, 1992.

[5] H. Jones, in *Computer Graphics.: Through Key Mathematics*, pp. 227, 2001.

[6] https://en.wikipedia.org/wiki/Constructive_solid_geometry

[7] http://glscene.sourceforge.net/oldsite/Gallery/boxedin_h.jpg

[8] http://ars.els-cdn.com/content/image/1-s2.0-S0010448508002030-gr3.jpg

[9] http://exocortex.com/products/implosia

[10] http://www.cs.cmu.edu/afs/cs/academic/class/15462-s13/www