Application of the Two-Sided Depth Test to CSG Rendering

Sudipto Guha
Univ. of Pennsylvania
sudipto@cis.upenn.edu

Shankar Krishnan
AT&T Labs–Research
krishnas@research.att.com

Kamesh Munagala*
Stanford University
kamesh@cs.stanford.edu

Suresh Venkatasubramanian
AT&T Labs–Research
suresh@research.att.com

Abstract

Shadow mapping is a technique for doing real-time shadowing. Recent work has shown that shadow mapping hardware can be used as a *second depth test* in addition to the z-test. In this paper, we explore the computational power provided by this second depth test by examining the problem of rendering objects described as CSG (Constructive Solid Geometry) expressions. We provide an algorithm that asymptotically improves the number of rendering passes required to display a CSG object by a factor of n by exploiting the two-sided depth test. Interestingly, a matching lower bound can be proved demonstrating that our algorithm is optimal.

Keywords: Constructive solid geometry, Graphics hardware, Z-buffer, Shadow mapping

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

1 Introduction

In recent years, the increased power of graphics rendering hardware has led to the use of the graphics pipeline for general purpose stream computations. One of the early examples of this was the use of hierarchical z-buffering for visibility calculations [Greene et al. 1993], and subsequently in programmable vertex shaders [Peercy et al. 2000; Lindholm et al. 2001; Proudfoot et al. 2001]. Other uses of the graphics pipeline as a general purpose stream computing engine have been demonstrated in computational geometry[Hoff III et al. 1999; Mustafa et al. 2001; Krishnan et al. 2002], robotics[Hoff et al. 2000], numerical analysis[Larsen and McAllister 2001], and ray tracing[Purcell et al. 2002].

In a recent development, work by Everitt *et al.* [2002] has shown that the shadow mapping hardware (supported in the nVidia GeForce3 and newer architectures) can be used to perform order-independent transparency. They demonstrate this by using the shadow mapping phase in the pipeline to filter out fragments that have a z-value less than (or greater than) values stored in a depth texture. This operation, combined with the standard z-test, provides a two-sided depth test on fragments. This feature is exploited in a technique they call *depth peeling* that can "peel" off layers of a scene one by one. Interestingly enough, the idea of using two-sided depth tests to implement depth peeling was proposed earlier by Mammen [1989], who used the idea of *virtual pixel maps*. The key observation by Everitt *et al.* was that existing shadow mapping hardware can be used to simulate this test.

*Supported by NSF grant CCR-0113217

Our Contributions In this paper, we study the computational power of the two-sided depth test in the context of rendering objects represented as CSG trees.

- We show that the two-sided depth test can be used to render CSG trees with a factor of n (number of primitives in the CSG expression) fewer passes than the best known OpenGL-based algorithms (see Section 2 for more details).
- Our algorithm can render arbitrary CSG objects, and does not require the explicit precalculation of levels that prior results did.
- We use no external storage or readbacks; all computations are performed directly on the GPU.
- Our algorithm works by performing a *topological sweep* over the arrangement of the objects; this technique may be of independent interest.

Paper Outline The rest of the paper is organized as follows. We discuss prior work in Section 2. We define the problem of rendering a CSG tree in Section 3, and present our solution for a single product in Section 4. The solution is extended to arbitrary CSG expressions in Section 5. We discuss implementation details and present our algorithm performance in Section 6 and we conclude in Section 7.

2 Prior Work

There has been extensive work on the problem of rendering solid objects defined in terms of CSG trees. A general survey of CSG methods is beyond the scope of this paper. We will focus solely on methods that make use of the graphics hardware.

Goldfeather *et al.* [1986] presented an algorithm for rendering a CSG tree of *convex* objects (and subsequently [1989] for non-convex objects) using an extension of the Pixel-Planes graphics hardware [Fuchs and Poulton 1981]. This algorithm was refined and implemented on modern graphics hardware by Wiegand [1996]. The running time of the algorithm, expressed as the number of rendering passes required, is essentially quadratic in the number of primitives (the running time also includes a quadratic term that depends on the *convexity* of the objects).

The Trickle algorithm, developed by Epstein *et al.* [1989] and later refined by Rossignac and Wu [1992], takes a different approach using "depth-interval buffers" (which essentially provide the functionality of a two-sided test) to do the rendering. Their approach requires three depth buffers, the two-sided depth test and two color buffers, and thus is not readily adaptable to current OpenGL-based architectures. Although they do not analyze their algorithm in terms of rendering passes, we believe that their approach (for each product) requires number of passes proportional to the depth complexity from the given viewpoint.

Stewart *et al.* [1998] presented an improvement to the Goldfeather *et al.*algorithm that takes into account the fact that objects may be disjoint and thus can be rendered in parallel. If the depth complexity of a collection of n primitives is k, then the modification proposed by Stewart *et al.*requires $O(kn)$ rendering passes. In the case when the primitives are sufficiently disjoint in screen space (and thus $k < n$), this algorithm is superior. Erhart and Tobler [2000] provide a modification to this algorithm that yields more accurate results (in terms of depth tests). However, in the worst case, their algorithm again requires $O(n^2)$ passes.

More recently, Stewart *et al.* [2000; 2002] present improvements that compute a CSG product in a constant number of passes when all the primitives are convex. They use a universal sequence to model the depth ordering of the primitives without having to compute an explicit front-to-back ordering. The caveat with this approach is that a quadratic number of objects are rendered in each pass (because primitives are duplicated).

All of the algorithms above compute a union of objects by merging the partial depth buffers obtained for each product. This merging step is performed via the use of readbacks, and is thus slow.

3 CSG Trees and Normalization

A three dimensional object can be described as the result of performing set operations ($\cup, \cap, \setminus$) on a ground set of shapes (called *primitives*). A *CSG tree* can be used to define an object by defining the sequence of operations that are performed.

The CSG tree is usually assumed to be in a canonical form to aid in rendering. A CSG tree is said to be in *sum-of-products* form if the expression it defines can be written as a union of intersections/subtractions (a sum of products). Such a tree is said to be *normalized*. Goldfeather *et al.* [1986] provide an algorithm for normalizing a CSG tree; we use their technique, and the rest of the paper assumes without loss of generality that the CSG tree has been normalized.

Given a normalized CSG tree and a procedure to compute the product of a set of primitives, unions can be computed easily by merging the results of individual products in the depth buffer. The above mentioned algorithms make use of this observation, and thus focus on the problem of rendering a CSG tree denoted by a single product. For clarity of presentation, we will explain the working of our algorithm on a single product, and subsequently we will show how the same ideas can be extended to render a sum of products.

3.1 Notation and Preliminaries

We denote a normalized CSG expression as $P_1 \cup \cdots \cup P_m$, where each P_i is a product of primitives. A single product is a general expression involving intersections and complementations. Consider a single product $P = (((\mathbf{o}_1 \cap \mathbf{o}_2) \setminus \mathbf{o}_3) \cap \mathbf{o}_4)$. P can be rewritten as $\mathbf{o}_1 \cap \mathbf{o}_2 \cap \overline{\mathbf{o}_3} \cap \mathbf{o}_4$. Thus each product is the intersection of a set of (possibly complemented) objects. For a product P, let $U(P)$ denote the set of uncomplemented objects and $C(P)$ denote the set of complemented objects. In this example, $U(P) = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_4\}$ and $C(P) = \{\mathbf{o}_3\}$.

Every object $\mathbf{o}$ is a collection of alternating front and back faces (or *layers* [Goldfeather et al. 1989]) as seen from the viewpoint. The *depth* $d(P)$ of a product P is the maximum number of layers in P (with respect to the viewpoint).

4 Rendering A Product

Consider a product P. Our algorithm works by traversing the layers of the primitives in P in a front-to-back order. It is easy to see that only the front faces of uncomplemented primitives and back faces of complemented primitives contribute to the visible portions of P. Hence, only these faces (termed *appropriate primitives*) are considered by our algorithm in the front-to-back traversal. Once a pixel in P is found, no further updates are made for this pixel. Thus the algorithm maintains a FRONT of all pixels (with associated depth values in the z-buffer) for which membership in P has not yet been determined. In each step,

1. We test which pixels in the FRONT satisfy membership in P.
2. If the pixel fails the membership test then its depth is updated to the depth of the next face in the front-to-back ordering – this is called *advancing the* FRONT .
3. If the pixel passes the membership test, a mask is applied to ensure its depth value is not updated in subsequent steps.

After the algorithm has traversed all layers the z-buffer holds the depth field of P. We then render all the objects with depth test set to `EQUAL` to obtain P. We illustrate the working of the algorithm in Figure 1.

Testing Product membership Assume that a point p is in the product and its depth is stored in the z-buffer. Goldfeather *et al.* [1986] made the observation that (a) if a primitive $\mathbf{o}$ occurs uncomplemented in a product, the number of layers of that primitive (both front and back) that have depth greater than p must be *odd*, and similarly (b) the number of layers (with depth greater than p) must be *even* if the object occurs complemented.

Let $f(\mathbf{o}, p)$ denote the number of front faces of $\mathbf{o}$ of depth greater than the depth of p. Similarly let $b(\mathbf{o}, p)$ denote the number of back faces of $\mathbf{o}$ satisfying the same depth condition. Since all objects are simple and thus have no self-intersections, each front face of $\mathbf{o}$ is followed immediately by a back face of $\mathbf{o}$, and thus $f(\mathbf{o}, p) \leq b(\mathbf{o}, p) \leq f(\mathbf{o}, p) + 1$. For an uncomplemented object, $b(\mathbf{o}, p) - f(\mathbf{o}, p) = 1$ and for a complemented object, $b(\mathbf{o}, p) - f(\mathbf{o}, p) = 0$. For a general product P, we can summarize the $|P|$ equations in a single condition as follows:

$$\sum_{\mathbf{o} \in C(P)} b(\mathbf{o}, p) - \sum_{\mathbf{o} \in C(P)} f(\mathbf{o}, p) + \sum_{\mathbf{o} \in U(P)} b(\mathbf{o}, p) - \sum_{\mathbf{o} \in U(P)} f(\mathbf{o}, p) = |U(P)|$$

It is not difficult to show that only points in the final product will satisfy the above equation. Moreover, this equation is crucial because it allows us to check membership for a point p in two rendering passes (instead of n). We use the stencil buffer to implement this test. We group together the back (front) faces of $U(P)$ and $C(P)$ in a single pass to increment (decrement) the stencil buffer. Pixels whose stencil value is $|U(P)|$ have passed the membership test, and are masked with a suitable value to prevent future depth updates.

Advancing the Front The FRONT is maintained as depth values in the z-buffer. The initial front is obtained by rendering the *appropriate primitives* with the depth-test set to `LESS`. To advance the front, we copy the depth buffer to a shadow buffer, and invoke the *depth peeling* subroutine to pass only those fragments whose depth is greater than the value in the shadow buffer using the alpha test. We refer the reader to [Everitt 2002] for details of the depth peeling algorithm. This test, coupled with the normal z-test (depth test set to `LESS`), provides fragments whose depth value is immediately behind the current front. In our case, we selectively advance the FRONT using the stencil mask. Observe the crucial role of the second depth test provided by the depth peeling routine. Without this, we would be unable to implement the two-sided depth test, $a \leq \texttt{Zvalue} < b$, and would not be able to advance the front in a single rendering pass.

The full algorithm is as follows. The sentinel N is used to mask points which have been determined to be in the product: the front is not updated for these pixels.

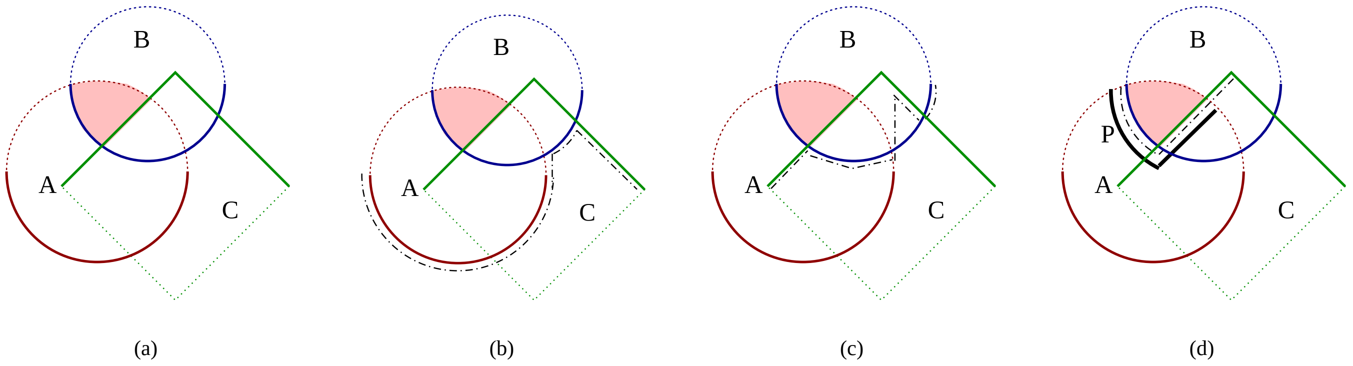


Figure 1: An illustration of the algorithm for a single product. (a) A product $P = A \cap B - C$. The original primitives are dotted and the appropriate primitives are drawn solid. P is shaded. (b) The initial contents of the front (dashed). (c) The front after a single step of the algorithm. (d) Final step: pixels marked solid are in P

Algorithm 1 Algorithm for a single product P

```
Initialize z-buffer to first front.
Initialize stencil buffer with 0
**while** stencil buffer contains 0 **do**
    **Test points on front for membership in** $P$
    Set stencil buffer to $N\ (> n)$ for pixels **passing** test
    Set stencil buffer to 0 for pixels **not passing** test
    **Advance front**
**end while**
```

5 Computing Unions of Products

We now describe the computation of a sum of products. Let the products be $P_1, P_2, \ldots P_m$. The computation proceeds incrementally. Assume that we have correctly computed $P_1 \cup \ldots \cup P_{i-1}$. At the start of the i^{th} step, the z-buffer contains the depth values for $P_1 \cup \ldots \cup P_{i-1}$ (denoted by D_{i-1}) and the color buffer contains the appropriate values (denoted by C_{i-1}).

We first copy the content of the z-buffer into a second shadow buffer *buf*. Then, we compute the depth field for P_i using the algorithm described in the previous section. This sets the color buffer appropriately as well. We now need to merge the two depth fields, retaining the minimum value at each pixel. Let the depth and color field for P_i be denoted by d_i and c_i respectively. Thus the new depth field $D_i = min(D_{i-1}, d_i)$. The new color field is

$$C_i = \begin{cases} C_{i-1} & \text{if } D_{i-1} < d_i, \\ c_i & \text{otherwise.} \end{cases}$$

This is accomplished in two phases. In the first phase, we identify those pixels where $d_i > D_{i-1}$ and tag them appropriately using the stencil buffer to be updated in the next phase. This is accomplished by setting the shadow test to pass fragments whose depth is *greater* than D_{i-1} and setting the depth test to EQUAL. The stencil function is set to clear the stencil bits if the depth test passes (i.e the fragment depth is equal to that in d_i). Intuitively, this encodes the two-sided test $D_{i-1} < d = d_i$, where d is the fragment depth. Now, rendering the faces of P_i has the effect of clearing the stencil buffer in all pixels for which the minimum depth is achieved by D_{i-1} i.e all pixels for which $D_i = D_{i-1}$. Note that this is precisely the set of pixels for which the current depth buffer contents are incorrect. The contents of the color buffer can be updated to reflect $P_1 \cup \ldots \cup P_i$ in this phase by going through one extra rendering of the faces of P_i at places where the stencil buffer is not cleared.

In the second phase, update the depth buffer to that in D_{i-1} wherever the stencil bits are cleared in the previous phase. We set the shadow test to pass fragments whose depth is *at most* ($\leq$) D_{i-1}. The depth test is set to GREATER. By rendering all objects in $P_1 \cup \ldots \cup P_{i-1}$, this two-sided depth test passes only fragments whose depth value is D_{i-1}. This completes the z-buffer update, and it now contains D_i. The union algorithm can be summarized as follows:

Algorithm 2 Algorithm for a union of products $P_1, \ldots, P_m$

```
Initialize shadow buffer *buf* to 1.
**for** $i$ = 1 to $m$ **do**
    Compute product $P_i$

    Set shadow test to **greater**
    Set depth test to **equal**
    Set stencil buffer to 0 on depth pass
    Render $P_i$ and update depth and color buffer

    Set shadow test to **less_or_equal**
    Set depth test to **greater**
    Set stencil test to **equal to** 0
    Render $P_1, \ldots, P_{i-1}$ and update depth buffer
    Copy depth buffer to shadow buffer *buf*
**end for**
```

Running Time Analysis The total number of rendering passes is the number of passes taken to compute each product, plus m passes to compute the union. Therefore the total number of passes is $m + 2\sum_i d(P_i) = O(\sum_i d(P_i))$. This running time is asymptotically superior to all prior techniques by a factor of n, where n is (on average) the number of primitives appearing in each product. Moreover, in our algorithm, we only render while there are pixels whose correct depth is yet to be determined. Therefore, in practice, our algorithm takes much fewer passes than predicted by the above worst-case expression. In contrast, the running times of the previous algorithms is "worst-case": the number of rendering passes required in any run is always the (same) worst-case.

6 Implementation Details

All our code was implemented using C++/OpenGL on a 1.8Ghz/1GB PC running Red Hat 7.3. The graphics card is an nVidia GeForce4 Ti4600. Our system performs no readbacks and uses no intermediate software buffers, while being able to handle *arbitrary* sums of products. For the purpose of this study, we considered four objects described by CSG trees: they are depicted in the right-most lines of Figure 2. In order to evaluate the performance of our scheme, we used two variants of our algorithm; **BASIC** is the algorithm described above, and **CONV** is a modification that processes convex objects more efficiently (however the overall algorithm structure remains the same). We compare both these methods to the algorithm used by Stewart *et al.* [2002], which we refer to as **SCS**. All running times are reported in frames per second. Table 1 summarizes the nature of the inputs and the running times obtained.

Object	#Products/	CSG Rendering		
	#Primitives	**BASIC**	**CONV**	**SCS**
GRID	1/26	26	57	32
HELIX	1/4	38	38	48^{**}
CUBE	2/4	7	21	1.23^{*}
HOLLOW	3/6	12	40	0.6^{**}

Table 1: Performance of our algorithms on some CSG models, in comparison to earlier work. Running times are reported in frames/second. An asterisk denotes artifacts in the solution and a double asterisk denotes an incorrect answer.

The table indicates two things: firstly that our algorithms (**BASIC,CONV**) obtain (overall) significantly better frame rates than **SCS**. Moreover, there are far fewer artifacts in our approach: this is possible due to the fewer number of EQUAL tests we perform in the z-buffer.

Figure 2 demonstrates the working of our system on the above models. For each object, the lefthand-most image displays all the primitive objects involved in the CSG operations. As we go from left to right, each image displays the portion of the final answer rendered at that layer. In the case of HOLLOW, the original CSG objects would not be visible in a direct superimposition, and so we render the set of primitive as two distinct figures (the two left-most ones) for ease of viewing. We also emphasize that we place *no* convexity restrictions on our primitives; HELIX contains nonconvex objects.

7 Conclusions

In this paper, we demonstrate that the two-sided depth test, as realized by using the shadow buffer, is a powerful operator in the graphics pipeline. We studied the specific problem of rendering objects represented as CSG trees and provided an algorithm that asymptotically improves the number of rendering passes by a factor of n. It is likely that there are many other problems for which specific aspects of the graphics hardware provides a tremendous advantage. In general, with the increasing power of graphics hardware, theoretical studies that attempt to ascertain the potential and the limits of this pipeline as a general purpose stream engine will be invaluable.

Acknowledgements We thank Nigel Stewart for making his implementation of the algorithm in [Stewart et al. 1998] available on his web page. We also thank nVidia for providing a sample implementation of depth peeling and helping us with the Linux implementation of it.

References

EPSTEIN, D., JANSEN, F., AND ROSSIGNAC, J. 1989. Z-buffer rendering from CSG: The Trickle algorithm. Research Report RC 15182, IBM.

ERHART, G., AND TOBLER, R. 2000. General purpose z-buffer CSG rendering with consumer level hardware. Tech. Rep. VRVis 003, VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH.

EVERITT, C., REGE, A., AND CEBENOYAN, C. 2002. Hardware shadow mapping. In *ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations using graphics hardware*, ACM, F38–F51.

EVERITT, C. 2002. Interactive order-independent transparency. Tech. rep., Nvidia Corporation. `http://developer.nvidia.com`.

FUCHS, H., AND POULTON, J. 1981. Pixel-planes: a VLSI-oriented design for 3-D raster graphics. *Proc. of the 7th Canadian Man-Computer Communications Conf.*, 343–347.

GOLDFEATHER, J., HULTQUIST, J. P. M., AND FUCHS, H. 1986. Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM Press, ACM, 107–116.

GOLDFEATHER, J., MOLNAR, S., TURK, G., AND FUCHS, H. 1989. Near realtime CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications 9*, 3 (May), 20–28.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-buffer visibility. *Computer Graphics 27*, Annual Conference Series, 231–238.

HOFF, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 2000. Interactive motion planning using hardware-accelerated computation of generalized Voronoi diagrams. In *Proc. IEEE International Conf. on Robotics and Automation*.

HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics 33*, Annual Conference Series, 277–286.

KRISHNAN, S., MUSTAFA, N., AND VENKATASUBRAMANIAN, S. 2002. Hardware-assisted computation of depth contours. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms*, 558–567.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiples using graphics hardware. In *Supercomputing*.

LINDHOLM, E., KILGARD, M., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proc. ACM SIGGRAPH 2001*.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (July), 43–55.

MUSTAFA, N., KOUTSOFIOS, E., KRISHNAN, S., AND VENKATASUBRAMANIAN, S. 2001. Hardware assisted view dependent map simplification. In *17th ACM Symposium on Computational Geometry*, 50–59.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proc. ACM SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 425–432.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proc. ACM SIGGRAPH 2001*.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 703–712.

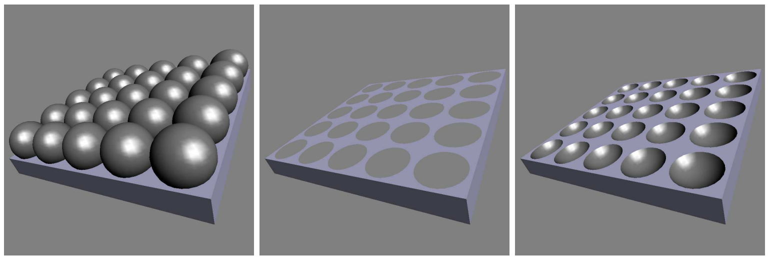
ROSSIGNAC, J., AND WU, J. 1992. Correct shading of regularized CSG solids using a depth-interval buffer. In *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, R. L. Grimsdale and A. Kaufman, Eds., Eurographics Seminars. Springer-Verlag, 117–138.

STEWART, N., LEACH, G., AND JOHN, S. 1998. An improved z-buffer CSG rendering algorithm. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, 25–30.

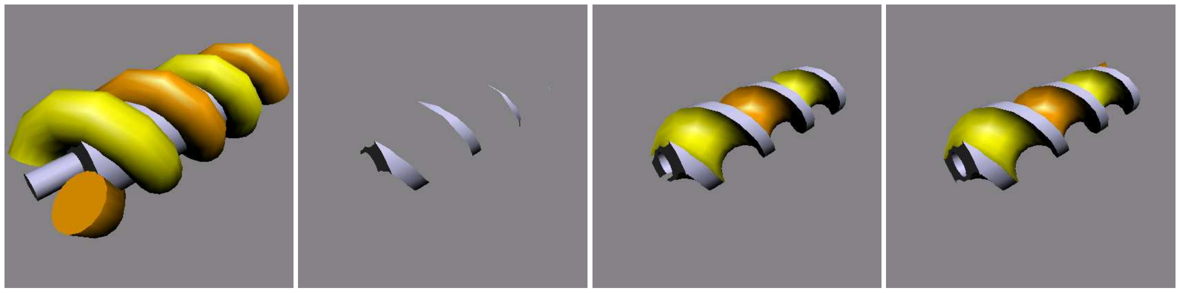
STEWART, N., LEACH, G., AND JOHN, S. 2000. A CSG rendering algorithm for convex objects. In *8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media - WSCG 2000*, vol. II, 369–372.

STEWART, N., LEACH, G., AND JOHN, S. 2002. Linear-time CSG rendering of intersected convex objects. In *10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2002*, vol. II, 437–444.

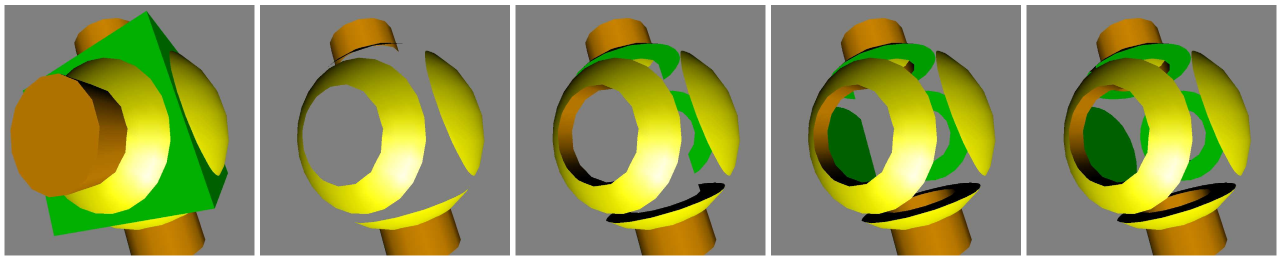
WIEGAND, T. F. 1996. Interactive rendering of CSG models. *Computer Graphics Forum 15*, 4, 249–261.



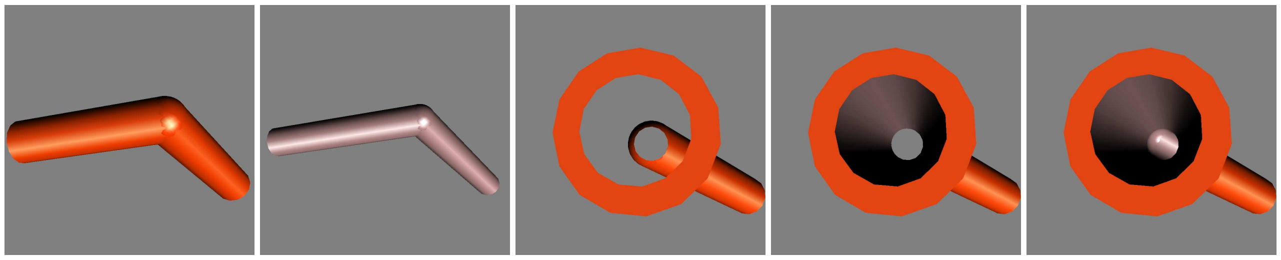
(a) GRID: The desired output is the flat sheet with the 25 spheres subtracted from it. Only two layers are necessary to compute the product.



(b) HELIX: Note that in this example two of the objects are non-convex (the two helices). The desired output is the subtraction of the two helices and the inner pipe from the outer pipe.



(c) CUBE: In this example, the boolean combination desired is the union of one of the cylinders with the product consisting the yellow sphere and the green cube minus the front-facing cylinder



(d) HOLLOW PIPE: In this example, for ease of viewing we show the original objects in the two left-most figures. The output should be a hollow pipe formed by the subtraction of the inner tube (colored in pink) from the outer tube (in red)

Figure 2: Examples of CSG renderings produced by our algorithm. In each example, the left-most figure depicts all the primitives prior to any boolean operations. Each subsequent figure depicts the rendered output after successive steps of the algorithm, and the right-most figure shows the final answer.