CSG Hidden Surface Algorithms for VLSI Hardware Systems

Frederik W. Jansen

Faculty of Industrial Design, Delft University of Technology
The Netherlands

Constructive Solid Geometry (CSG) is a solid modeling representation that defines objects as Boolean combinations of primitive solids. For the display of such objects, both the visibility problem and the problem of combining the primitive solids into one composite object have to be solved. Recently, several CSG hidden surface algorithms have been published that reduce these two problems to a combination of simple depth comparisons and logical operations at the pixel level that can be performed in VLSI hardware display systems. An overview of these algorithms is given. Furthermore, a CSG depth-buffer algorithm is presented that combines these algorithms.

1. Introduction

Projective display of three dimensional objects involves a sequence of transformations, clipping and scanning operations known as the 'viewing pipeline' [Foley and van Dam, 1982]. Parts of this viewing pipeline have been implemented in VLSI hardware systems to obtain nearly realtime performances [Clark, 1982]. Most of these systems are based on the depth-buffer (z-buffer) algorithm because the depth sort at the pixel level fits best in a pipe-line approach. A drawback of these systems is that they can only display objects defined with a boundary representation and not objects defined as a combination of elementary volumes (Constructive Solid Geometry).

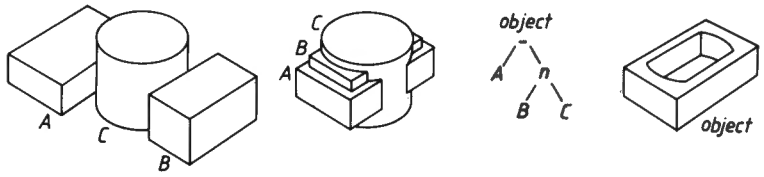
An interesting development are the so-called logic enhanced frame buffer systems, such as the Pixel-planes and Pixel-powers systems developed at the University of North Carolina [Fuchs et al, 1985], [Goldfeather and Fuchs, 1986a]. These systems incorporate an arithmetic preprocessor that evaluates a linear, resp. quadratic expression for each pixel in parallel, and a frame buffer that has at each pixel a one-bit processor with associated image-memory. With linear resp. quadratic expressions as input, and the ability to perform bitwise operations at the pixel-level, a large number of complicated rendering and shading calculations can be performed for polygon models and quadratic halfspace models. Related to these systems are the scan-line systems [Demetrescu, 1985], [Gharachorloo and Pottle, 1985]. Instead of a complete frame-buffer, these systems only contain a buffer for one scan-line, and every scan-line of the image is generated from this buffer within the video refresh rate.

The ability to perform simple logic operations at the pixel level makes it possible to use these systems for the display of objects defined with Constructive Solid Geometry [Goldfeather et al., 1986b], [Jansen, 1986]. Earlier, Thomas [1983] already showed the feasibility of such a CSG algorithm for VLSI implementation.

In this article, a short overview of these CSG algorithms is given. After an introduction on CSG evaluation in section 2, a depth-buffer algorithm is described in section 3. Tree restructuring and tree pruning methods, which reduce memory use and improve the efficiency, are the subject of sections 4 and 5. In section 6, the CSG depth-buffer algorithm is extended to include the tree restructuring and tree pruning methods.

2. CSG evaluation

With Constructive Solid Geometry, objects are defined as Boolean combinations of primitive solids, such as block, sphere, cylinder and cone. The CSG object can be represented as a binary tree (see fig. 1), with the primitives at the leaf nodes and the set operators at the composite nodes.

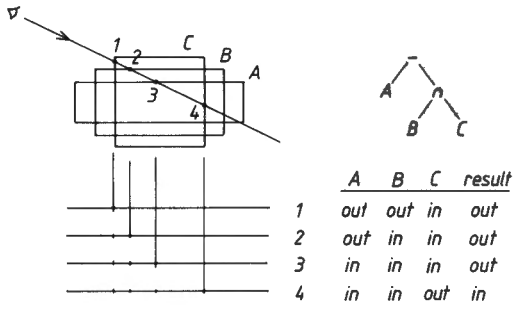


a. primitives b. positioned c. CSG-tree d. result

Figure 1. CSG representation

To display objects so defined, both the visibility problem and the problem of combining the primitive objects into one object have to be solved. This requires both an evaluation of the surface elements of the primitives to test whether a surface element contributes to the boundary of the resulting composite object, and a depth sort to test whether a surface element will be visible or will be hidden.

Every point in space can be classified as 'in' or 'out' of a CSG object by classifying the point with respect to the primitive solids and combining the individual point-primitive classifications according to the set operations in the CSG tree. A point on a projection line (through a pixel and the eyepoint) can be classified with respect to a primitive solid by testing (with depth comparisons) whether the point is 'in' the primitive or 'out' the primitive. A point is 'in' (a convex primitive) if it is both behind (or on) the forward facing boundary and in front of (or on) the backward facing boundary of the primitive (seen from the eye point). Traversing the CSG tree in post-order, for each leaf node the point is classified as 'in' or 'out' with respect to the primitive solid at that leaf node, and at each composite node the left and right classifications are combined according to the Boolean operators at that composite node. The classification at the root gives the 'in' or 'out' classification of the point with respect to the resulting composite object (see fig. 2).



	A	B	C	result
1	out	out	in	out
2	out	in	in	out
3	in	in	in	out
4	in	in	out	in

Figure 2. CSG evaluation

In a CSG hidden surface algorithm, the CSG point classification can be applied for the evaluation of the boundary elements of the primitives. A surface element that, for a pixel, classifies as 'in' and is found to be visible (nearer in depth than the other valid elements), can be rendered for that pixel.

In software implementations, the intersection or scanning of the surface elements is the crucial time-consuming activity. The depth value of a surface element for a pixel is therefore calculated once, and stored for further processing. This includes a depth sort and an evaluation in depth order until a valid and thus visible element is found [Atherton, 1983]. In the logic enhanced frame buffer systems, the reverse is true. The scanning is relatively cheap and the memory capacity is the limiting factor. Therefore, instead of storing the depth information, it is preferable to recalculate this information by scanning the primitives as often as necessary.

3. A CSG hidden surface algorithm

This leads to an algorithm [Jansen, 1986] that first evaluates the surface elements by depth comparisons with the other primitives, and then renders the surface element with a depth-buffer (or z-buffer) algorithm into the frame buffer. This algorithm requires two depth-buffers. The first buffer is used for the depth comparisons of the CSG evaluation, and the second buffer is used for the standard depth-buffer algorithm.

The processing for each pixel (in parallel) is as follows. For a primitive the forward facing surface elements are scanned into the first depth-buffer. The surface elements are evaluated by traversing the CSG tree and scanning for each leaf node the forward and backward facing part of the boundary of the primitive at that node for comparison with the depth value of the test primitive. This classification is done for each pixel independently and in parallel. The result is stored as '0' or '1' in the pixel memory. At each composite node two bits of the stack are combined according to the set operation of that node (see fig. 3). If the depth of the CSG tree is n, then n-1 bits are needed to serve as a stack to store the intermediate results.

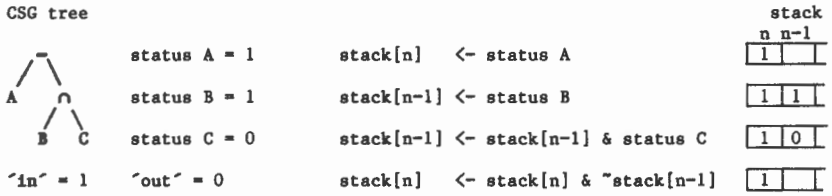


Figure 3. CSG tree evaluation with bitwise operations
(& = Boolean intersection, ~ = one's complement)

When a primitive boundary is found to be valid for a pixel, the primitive is rendered into the frame buffer for that pixel with the standard depth-buffer algorithm. This is repeated until all the primitives are processed.

For the details of the scanning algorithm see [Jansen, 1986], where also a depth-sort version of this algorithm is described that evaluates the primitives in depth-order.

4. Tree restructuring

A drawback of the just described algorithms is the need to reserve a certain number of bits of the image memory for the stack of the CSG tree traversal. This can be overcome by a restructuring of the CSG-tree [Thomas, 1983], [Okino et al., 1984], [Goldfeather et al, 1986b].

Any CSG tree can be converted into a sum of product form (union of intersection form), that is a union of subtrees with only Boolean intersection operations,

$$\bigcup_i \bigcap_j P_{ij}$$

by the following substitutions:

$$A - B = A \cap \overline{B}$$

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

$$P = \begin{cases} \text{primitive at a leaf node} \\ \text{complement of primitive at a complemented leaf node} \end{cases}$$

The display operation is denoted by D(object), and for projective display only the part of the boundary of a primitive that is directed towards the view point can be visible.

$$D(P) = \begin{cases} D(Pf) & \text{forward facing} \quad \text{if P is primitive} \\ D(Pb) & \text{backward facing} \quad \text{if P is complement of primitive} \end{cases}$$

To render the union of products, it suffices to render the products with a depth buffer algorithm.

$$D(\text{object}) = D(\bigcup_i \bigcap_{j=1}^{m(i)} P_{ij}) = \bigcup_i D(\bigcap_{j=1}^{m(i)} P_{ij}) = \min_i D(\bigcap_{j=1}^{m(i)} P_{ij})$$

Each product can be rendered by displaying the valid parts of the primitive boundaries, i.e. those parts that are contained in the product $\bigcap P_{ij}$.

$$D(\bigcap_{j=1}^{m(i)} P_{ij}) = \bigcup_{k=1}^{m(i)} (D(P_{ik}) \bigcap_{j=1,\, j\neq k}^{m(i)} P_{ij})$$

For each primitive, the contribution is the sum of the contributions of all its products, and thus for the total object the rendering is equal to:

$$D(\text{object}) = \bigcup_{i}^{n(i)} \bigcup_{j=1} \left(D(P_i) \cap \bigcap_{k=1,k\neq i}^{m(j)} P_{jk} \right) = \min_{i}^{n(i)} \min_{j=1} \left(D(P_i) \cap \bigcap_{k=1,\, k\neq i}^{m(j)} P_{jk} \right)$$

where i is the index for the primitives, n(i) is the number of products for primitive i (the number of products that contain the primitive i), and m(j) is the number of primitives contained in the product j. Each primitive can be rendered by scanning D(P) for each product, and disabling those parts that are not contained in the product. For a detailed description of possible implementations see [Thomas, 1983] and [Goldfeather et al., 1986b].

The tree restructuring algorithm eliminates the evaluation tests at the cost of rendering the primitives multiple times. The size of the sum of products form depends largely on the number of Boolean intersection and difference operators in the top part of the tree (the worst case is a tree of intersection operations with pairwise unioned primitives at the leaves; this gives 2**2(n-1) terms for 2**n primitives). In practice, however, union operators dominate at the top of a CSG tree, because, in general, larger objects are assemblies (unions) of smaller parts.

5. Tree pruning

The performance can be improved by avoiding empty products. A product contributes to the rendering of P, if all primitives in the product at least partly overlap with P. This means that the rendering of P is dependent on those primitives that share a Boolean intersection operation with P in the original tree and (at least partially) overlap with P (Boolean difference operations are assumed to be converted to Boolean intersection operations by complementing the right node). The primitives that satisfy these requirements will be called active primitives for P.

To eliminate empty product forms, for each primitive P a new CSG tree is calculated in a preprocessing step that only contains the active primitives. All non-active primitives are pruned from the tree according to the following pruning rules:

$$\text{out} \cup \text{tree} = \text{tree}$$
$$\text{out} \cap \text{tree} = \text{out}$$

The resulting tree is the active CSG tree for P. Restructuring the total CSG tree on basis of the active trees leads to

$$D(\text{object}) = \bigcup_{i}^{n(i)} \bigcup_{j=1} \left(D(P_i) \cap \bigcap_{k=1,k\neq i}^{m(j)} P'_{jk} \right) = \min_{i}^{n(i)} \min_{j=1} \left(D(P_i) \cap \bigcap_{k=1,k\neq i}^{m(j)} P'_{jk} \right)$$

Where $\bigcap P'$ denotes a product resulting from the active CSG tree of P.

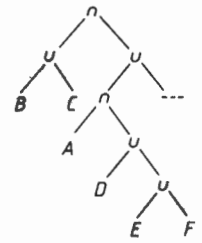
Experiments with realistic models will have to show whether the use of active trees leads to linear results. This implies that for complex models the average number of active primitives may be dependent to a certain extent on the number of Boolean intersection and difference operations in the CSG tree, but will be largely independent of the total number of primitives.

Improvements may also be expected from a spatial subdivision strategy, for instance a recursive binary subdivision of the image, alternately in the x and y direction, with associated tree pruning [Woodwark and Quinlan, 1982], [Thomas, 1983].

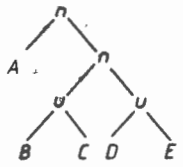
6. A CSG depth-buffer algorithm

In this section, a new version of the CSG depth-buffer algorithm will be derived that integrates the tree restructuring and tree pruning of sections 4 and 5 with the tree traversal method described in section 3. The basic idea is that the tree restructuring (and the corresponding tree expansion) can be simulated by additional tree traversals.

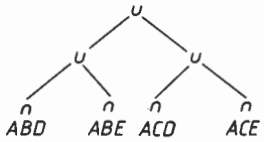
To clarify the concept, the following example will be used. Figure 4a shows a CSG-tree. From this tree an active tree is derived for the primitive A (fig. 4b). Tree restructuring gives the sum of products form (fig. 4c). The primitive A can now be rendered by scanning, with a depth-buffer test, those parts of the boundary of A that are inside the different products (fig. 4d).



a. CSG tree



b. Active tree for A
(Primitive F is removed because it did not intersect A)



c. Sum of products for A

$$D(A) = (D(A_f) \cap BD) \cup (D(A_f) \cap BE) \cup (D(A_f) \cap CD) \cup (D(A_f) \cap CE)$$

d. Scanning for primitive A

Figure 4. Example of tree restructuring.

Instead of restructuring the tree, each primitive combination (product) can also be found by a specific traversal through the tree: each product represents a unique combination of sub-products of the union operations. The tree traversal is guided by the following rules:

- At each composite node with a Boolean union operation, only one of the subtrees is visited,
- At each composite node with a Boolean intersection operation both subtrees are visited,
- At each node, a Boolean variable (false/true) records the status information that indicates which subtree still has to be visited.

If the tree (fig. 4b) is traversed in post order, the order of the primitives in the product is the same as indicated in fig. 4d.

Initially, the status of all nodes is cleared. The tree traversal for the first product starts with descending the left most branch of the tree. After processing leafnode B, the tree is ascended (for the union operations only one subtree is processed at a time), and the right subtree of the Boolean intersection operation is descended. Next primitive D is processed and marked (the processing refers to

the scanning operations described in section 3). In the next traversal, B and E are processed. Now the composite node of D and E can be set (If a composite node is set, the status variables of its subnodes can be cleared). For B now all combinations have been processed, B is set and the status information of the right subnodes (in this case the composite node of D and E) is cleared. In the next tree traversal, C is visited and the same combinations are found for C. After processing these combinations, both the left and right subtrees are set and all products are found.

The algorithm combines the tree traversal algorithm (section 3) with the tree restructuring of (section 4), in the sense that scanning is minimized when stack-bits can be used for the tree evaluation (section 2), and memory use is minimized (minimal 3 bits) with the tree restructuring or the tree traversal method described in this section. Also a combination of both methods can be used: the top part of the tree is processed with a stack and subtrees are processed with a repeated tree traversal. If the number of available memory bits is m, and the depth of the tree is n then:
- for $m = 3$, the tree is traversed for every product,
- for $3 < m < n$, $m-3$ bits are used for the first $(m-3)$ levels of the tree,
- for $m = n$, the tree is traversed only once for a primitive.

Any desired trade-off between speed and memory can be chosen. Minimal memory use may be desired for frame buffer systems. Optimal speed may be desired for scan-line systems. The tree traversal method described may be used in that case to prevent overflow. Choices for best combinations of algorithms and hardware systems will have to be verified by experiments with realistic models.

7. Conclusions

Fast rendering of linear and quadratic surfaces with VLSI hardware systems has given rise to a new generation of CSG hidden surface algorithms, based on tree restructuring and tree pruning techniques. An overview of these algorithms, and a CSG depth-buffer algorithm has been described that integrates the tree restructuring and tree pruning techniques into one tree traversal method.

Acknowledgements

I would like to thank Petri Koistinen (Helsinki University of Technology) for earlier discussions on sum of product forms. I also thank Jarek Rossignac (IBM Thomas J. Watson Research Center) for discussions on active trees, and Jack van Wijk and Wim Bronsvoort for their comments on an earlier version of this paper.

References

Atherton, P.R. (1983), A scan-line hidden surface removal procedure for constructive solid geometry, Computer Graphics 17(1983)3, 73-82, Siggraph83.

Clark, J. (1982), The geometric engine: a VLSI geometry system for graphics, Computer Graphics 16(1982)3, 127-133, Siggraph82.

Demetrescu, S. (1985), High speed image rasterization using scan-line access memories, Proc. 1985 Chapel Hill conference on very large scale integration, Computer Science Press, Inc.

Foley, J.D., van Dam, A. (1982), Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, 1982.

Fuchs, H., Goldfeather, J., Hultquist, J.P., Spach, S., Austin, J.D., Brooks Jr., F.P., Eyles, J.G., Poulton, J. (1985), Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes, Computer Graphics 19(1985)3, 111-120, Siggraph85.

Gharachorloo, N., Pottle, C. (1985), Super buffer: a systolic VLSI graphics engine for real time raster image generation, Proc. 1985 Chapel Hill conference on very large scale integration, 33-44, Computer Science Press, Inc.

Goldfeather, J., Fuchs, H. (1986a), Quadratic surface rendering on a logic-enhanced frame-buffer memory, IEEE Computer Graphics and Applications 6(1986)1, 48-58.

Goldfeather, J., Hultquist, J.P.M, Fuchs, H. (1986b), Fast constructive solid geometry in the pixel-powers graphics system, Computer Graphics 20(1986)4, 107-116, Siggraph86.

Jansen, F.W. (1986), A pixel-parallel hidden surface algorithm for constructive solid geometry, Proceedings Eurographics´86, North Holland, 1986,

Okino, N., Kakazu, Y., Morimoto, M. (1984), Extended depth-buffer algorithms for hidden-surface visualization, IEEE Computer Graphics and Applications 4(1984)5, 79-88.

Shiroma, Y., Okino, N., Kakazu, Y. (1982), Research on 3-D geometric modeling by sweep primitives, Proceedings CAD´82 conference, 671-680.

Thomas, A.L. (1983), Geometric modeling and display primitives towards special-ised hardware, Computer Graphics 17(1983)3, 299-310, Siggraph83.

Woodwark, J.R., Quinlan, K.M. (1982), Reducing the effect of complexity on volume model evaluation, CAD 14(1982)2, 89-95.