Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes

John Hable **Jarek Rossignac**
Georgia Institute of Technology, GVU Center

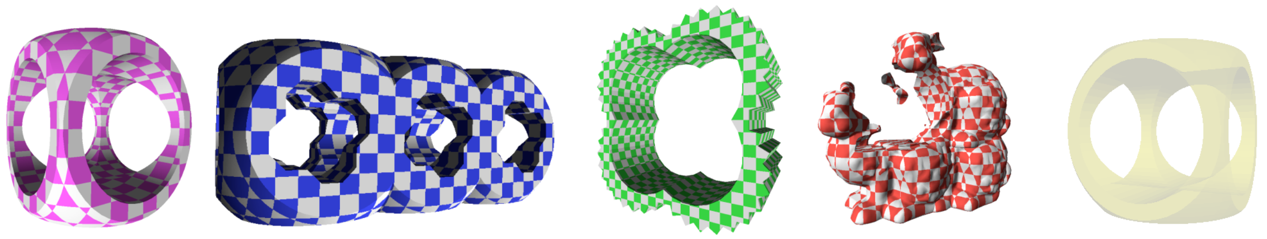

Figure 0: Solids X_1, X_2, X_3, X_4, and X_5 (from left) were rendered with Blister in real-time directly from their CSG expressions. X_3 has 48 CSG primitives. Its disjunctive form has over 30,000 products. X_4 subtracts a bunny from the union of 3 other bunnies. X_5 is transparent.

Abstract
By combining depth-peeling with a linear formulation of a Boolean expression called Blist, the Blister algorithm renders an arbitrary CSG model of n primitives in at most k steps, where k is the number of depth-layers in the arrangement of the primitives. Each step starts by rendering each primitive to produce candidate surfels on the next depth-layer. Then, it renders the primitives again, one at a time, to classify the candidate surfels against the primitive and to evaluate the Boolean expression directly on the GPU. Since Blist does not expand the CSG expression into a disjunctive (sum-of-products) form, Blister has O(kn) time complexity. We explain the Blist formulation while providing algorithms for CSG-to-Blist conversion and Blist-based parallel surfel classification. We report real-time performance for non-trivial CSG models. On hardware with an 8-bit stencil buffer, we can render all possible CSG expressions with 3909 primitives.

Keywords: CSG, Graphics hardware, Depth peeling, GPU

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

1 Introduction

Designers of manufactured products and processes or of videogame environments use CSG (Constructive Solid Geometry) [Requicha 1980; Hoffmann 1989] for specifying solid models as Boolean combinations (Fig. 1) of primitives. CSG expressions can be converted to a polygonal mesh through boundary evaluation, but conversion algorithms remain too slow for real-time graphic [Requicha and Voelcker 1985; Tawfik 1991; Keyser et al. 2002, Banerjee et al. 1993]. Consequently, many algorithms exist for rendering images of a CSG solid from a specific viewpoint without explicit calculation of the boundary.

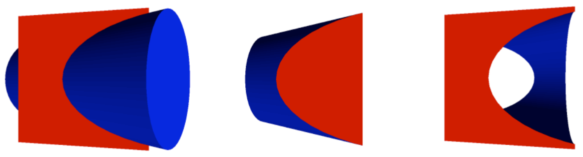

Figure 1: Boolean combinations of a red block A with a blue cylinder B: union A∪B, intersection A∩B, and difference A–B.

A CSG expression describes a tree, in which leaves represent primitives and nodes represent Boolean operators (Fig. 2). For simplicity, we assume that parent-nodes with more than two children have been expanded into a binary tree.

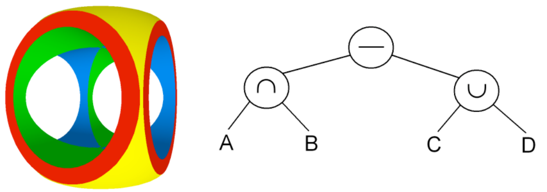

Figure 2: A CSG solid (left) and its CSG tree (right).

Because industrial and architectural CAD models may contain hundreds or even thousands of primitives, there is a need for improving the space-and-time complexity of CSG-rendering algorithms. Most previously proposed hardware-assisted algorithms expand the CSG expression into a sum-of-products (**disjunctive form**) [Epstein et al. 1989; Rossignac 1994; Goldfeather et al. 1986; Goldfeather et al. 1989]. This form often contains an exponential number of products, hence drastically reducing performance for complex CSG models. Other algorithms partially evaluate the model [Rappoport and Spitz 1997] or require expensive custom hardware [Ellis et al. 1991].

The method introduced here converts an arbitrary Boolean combination of primitives into its **Blist** form [Rossignac 1999]—a list of the original primitives, each represented only once. Our **Blister** algorithm, which stands for Blist-Expression Renderer, uses the Blist formulation to render the corresponding solid on commodity graphics hardware.

With only trivial pre-computation, Blister renders CSG models of 20 primitives at 11 frames per second on a nVidia GeForce 6800 adapter, and models with 48 primitives at 2 frames per second. With only 8 available stencil planes, we can guarantee support for all CSG expressions with less than 3909 primitives, while usually handling much larger expressions. Since Blister generates and classifies the surfels on primitive boundaries in front-to-back order through a modified version of depth peeling [Everitt 2002], we trivially extend Blister to the rendering of CSG scenes that contain semi-transparent primitives. We also show that Blister may be used to produce images with shadows directly from CSG.

2 Background and Prior Art

Assume that the intersection of ray R_i from the viewpoint through pixel i with primitive P_j, has k_{ij} disjoint segments. Let k_i be the sum of k_{ij} for all j. The **depth complexity** k of an arrangement of primitives is the maximum of k_i over all pixels i. Note that k is view dependent.

We assume without loss of generality that all CSG expressions have been converted into their **positive form**. Let B' denote the complement of primitive B and assume that the ' operator has highest priority in CSG expressions. First, we convert all differences into intersections with a complemented right argument ($A–B \Rightarrow A \cap B'$). Then, we propagate the complement down the tree applying DeMorgan's laws: $(A \cup B)' \Rightarrow (A' \cap B')$ and $(A \cap B)' \Rightarrow (A' \cup B')$ and $(A'' \Rightarrow A)$. For example, consider $X=(((A \cap B) \cap C) \cup D)-((E \cup F) \cap G)$ shown in Fig. 3. Its positive form is $Y=(((A \cap B) \cap C) \cup D) \cap ((E' \cap F') \cup G')$. Primitives that appear complemented in the positive form (E', F', and G') are said to be **negative**, while the remaining primitives are **positive**.

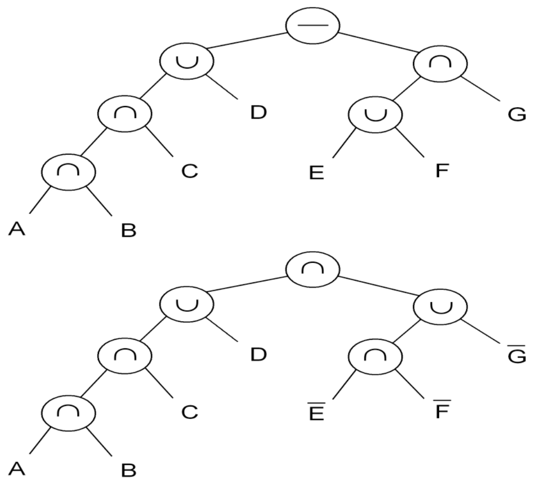

Figure 3: The CSG tree (top) for X= (ABC∪D)–(E∪F)G and for its Positive form (bottom) Y=(ABC∪D)(E'F'∪G').

For conciseness, we assume that ∩ operations have higher priority than ∪ and we often omit the ∩ symbols. Assuming left-to-right evaluation, Y may be written as (ABC∪D)(E'F'∪G'). We also use the symbols *T* and *F* to denote an expression that has been evaluated to *True* and *False* respectively.

To support hardware acceleration, several techniques convert CSG expression into their **disjunctive form**, which is a union of products, where each product is the intersection of positive and negative primitives [Goldfeather et al. 1986; Epstein et al. 1998]. This conversion may produce a number of products that grows exponentially with the number of primitives in the CSG ecpression (Fig. 4). Some of the products that define null objects may be pruned [Tilove 1984].

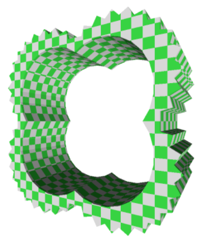

Figure 4: This solid was defined in CSG with 48 primitives. Its disjunctive form has 31104 products with 5 primitives per product. Every primitive intersects every other primitive, so no products can be pruned. The Blister algorithm renders this solid at interactive frame-rates.

The boundary of a CSG solid is a subset of the union of the boundaries of its primitives [Tilove 1984]. Consequently, many CSG rendering approaches follow variations of three basic steps:

1. Generate **candidate surfels** (points where the rays from the viewpoint through pixels intersect a primitive's boundary).
2. **Classify** the candidate surfels against the CSG expression to reject those not on the CSG solid.
3. For each pixel, select the surfel **closest** to the viewpoint.

Step 1 is typically performed by a rasterizer. When the depth complexity of a primitive exceeds 1, several passes and a **counting** or a **peeling** mechanism are used to ensure that all layers are sampled. Goldfeather et al. [1986] allocated several stencil bits to **count** how many times a pixel has been intersected during the rasterization of a given primitive and to lock the surfel produced during the i^{th} hit of the i^{th} raster pass. This approach was implemented on Pixel Planes [Fuchs and Poulton 1981] and later on commodity graphics hardware [Wiegand 1996].

In contrast, **Peeling** (Fig. 5) produces layers in depth order. The Trickle algorithm [Epstein et al. 1989] uses a **Depth-Interval Buffer** to build the layers in depth-order for each product of the disjunctive form. Assume that the depth of the previous layer is stored in z-buffer F (front) and that z-buffer B (back) is initialized to infinity. The primitives of the product are rasterized. Each time we find a surfel whose depth Z falls between the values F and B at the corresponding pixel, we replace B by Z and save the color of that surfel in the color buffer. The process produces the front-most layer behind F (i.e., the next peel). This algorithm may be applied to any arbitrary arrangement of primitives, not necessarily in a product. Because a Depth Interval Buffer was not supported in hardware, the Trickle algorithm was implemented [Rossignac and Wu 1992] using a software comparison of depth values stored in the hardware z-buffer with z-values saved in memory. Everitt [2002] implemented depth-peeling in hardware based on Mamman's design [1989].

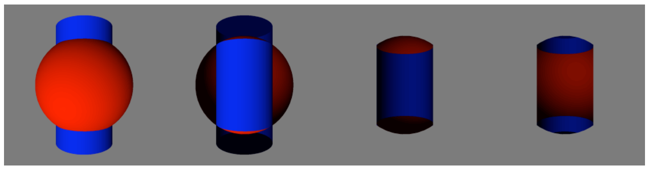

Figure 5: From left to right, we show the 4 depth-layers of a scene with a sphere and a cylinder. For better visibility, the front faces are shaded with lighter colors.

Step 3 can be supported through the use of a z-buffer hardware for combining: (1) software classified surfels [Rossignac and Requicha 1986], (2) visible parts of each product [Epstein et al.

1992], or (3) the contribution of each layer [Goldfeather et al. 1986].

Step 2 may be implemented by storing in system memory the list of all surfels on the primitives, followed by classifying them in order against the primitives of the CSG tree and combining the results of this classification in software. This procedure was the essence of the early approach of Rossignac and Requicha [1986]. Jansen [1986] proposed a hardware implementation of the classification for Pixel Planes that would store surfel-primitive classification results as stencil bits for each pixel and then perform parallel logical combinations on these bits according to the CSG expression. For Boolean operations on sculptured primitives, Adams and Dutré [2003] use an oversampled surfel approximation of each solid: the surfels of one argument are tested against an octree representation of the other and vice versa. Instead of surfels, the idea may be applied to interior samples using 3D textures [Chen and Fang 1999; Fang and Liao 2000, Liao and Fang 2002]. These approaches are particularly attractive for interactively viewing constant CSG scenes, since they pre-compute surfel classifications. While faster than boundary evaluation, the cost of surfel classification for complex CSG expressions may limit the applicability of these approaches to simple CSG models.

Accuracy and performance may be increased if, instead of storing a sampling of surfels, the surfels are recomputed for each frame by rasterization at the desired resolution and then classified using graphics hardware. Hence, we will now review Step 2 approaches implemented by first rasterizing the primitives and storing a layer of surfels in a temporary color and z-buffer. To classify the surfels of a layer, each primitive is rasterized while toggling a parity bit at each pixel where the primitive's boundary occludes the corresponding surfel, as used in [Rossignac et al. 1992]. The value of the parity bit indicates whether the pixel lies inside or outside of the primitive (Fig. 6).

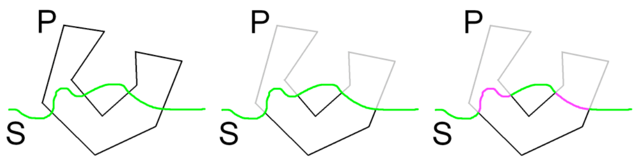

Figure 6: The surfels on the green layer are tested against the grey primitive P (left). Portions of P in front of the green layer (center) toggle the parity bits of the pixels they cover. Layer surfels with odd parity are inside P (right).

These surfel/primitive classification results must be merged according to the CSG expression. To avoid the need of implementing a stack for each pixel [Jansen 1986], the CSG expression is often converted into a disjunctive form [Goldfeather et al. 1986; Epstein et al. 1989]. Classifying these surfels of a layer against the product of the disjunctive form requires a single bit of the stencil for a parity test on that surfel. Another bit records the combined outcome of the classification against all previously scanned primitives in a product. If the surfel is found to be outside of a positive primitive or inside a complemented primitive, that surfel is outside of the product.

Several variations have been proposed to accelerate the rendering of a product for the general case [Guha et al. 2003, Stewart et al. 1998; Erhart and Tobler 2000], for the case where all primitives are convex [Jansen 1986; Stewart et al. 2000; Stewart et al. 2002], and for products with small overlap graph subtraction sequences [Stewart 2003]. The Trickle algorithm [Epstein et al. 1996] peels the layers of a general product in depth order, while at the same time performing classification. Through simple graph rewriting, Rappoport and Spitz [1997] convert the CSG expression into a CDA (Convex Difference Aggregate). This CDA is the union of products, each being the difference between a convex polyhedron P and the union of convex polyhedral holes contained in P. They also pre-compute the intersections of all pairs of convex polyhedral holes in a product using QuickHull [Barber at al. 1993]. Finally, Kelley et al. [1994] designed custom graphics hardware to render small CSG expressions at interactive frame-rates while supporting transparency.

Blister avoids the need for converting the CSG model into a disjunctive or CDA form and does not require storing the results of multiple surfel/primitive classifications at each pixel. It peels the entire arrangement of primitives in depth order. Each peel is classified according to its CSG expression and then combined. This operation is shown in Fig. 7. The classification step is made possible through the novel Blist formulation. Peeling stops when all pixels are behind a successfully classified completely opaque surfel or fall outside a pre-computed superset of the CSG solid. For a scene with k depth layers and n primitives, Blister runs in O(kn) time.

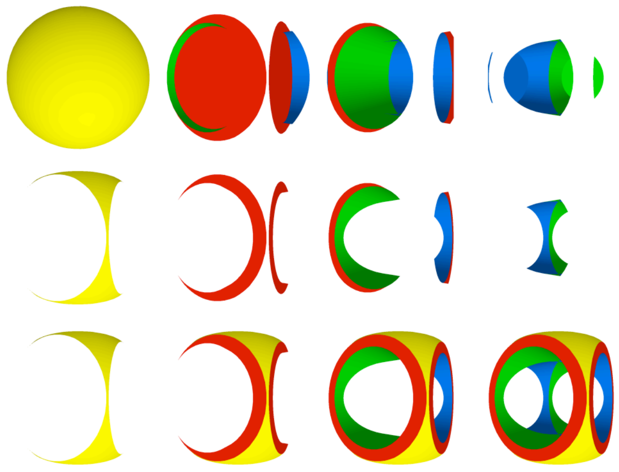
Figure 7: In the first row, object X_1 from Fig. 0 is decomposed into 4 layers. The second shows the surfels that pass the Blist evaluation, and the third row shows the combined final image.

3 Blist

To understand Blist, assume that we are going to classify a surfel *s* at a particular pixel *p* against a CSG expression Y, which is in positive-form. Also, assume that *s* is **not on the boundary** of any primitive. We will later remove this restriction.

Note that surfels of the same layer at other pixels will be classified in parallel against Y. We rasterize the primitives in the order (A, B, C… N) of the CSG expression Y to determine (by counting parity as detailed below) whether *s* lies inside or outside each primitive. After testing *s* against a primitive, we update the **status** of *s* (stored in the **stencil** bits at the corresponding pixel) to reflect what we have learned about the classification of *s* against Y thus far. Once all the primitives have been scanned, the status of *s* will indicate whether it lies inside or outside of Y. The rest of this section discusses what we store as status for each pixel and how many bits are required to keep track of the status at each pixel.

3.1 Blist Fundamentals

Consider that surfel *s* has been classified with respect to the left most node P of the simple CSG expression P∗N, where ∗ is either an intersection or union operator and where N is either a primitive or a sub-expression. We can replace P with *T* (*True*) if *s* is in P or with *F* (*False*) otherwise. Now we can simplify the expression. We have 4 cases, depending whether P is *T* or *F* and whether ∗ is ∪ or ∩. The simplification rules are: *T*∩N⇒N, *F*∪N⇒N, *F*∩N⇒*F*, and *T*∪N⇒*T*.

Assume now that P is the left-most primitive in a positive form expression Y and that Q is the next primitive in Y. P and Q may be the two arguments of the same operator, as in Y=(PQ∪C)D, where N is Q. However, Q may also be the left most node of a sub-expression N that is the right sibling of P, as in Y=(P∪QC)D, where N is QC.

When one of the first two simplification rules is applied (*T*∩N⇒N or *F*∪N⇒N), Q becomes the left-most symbol of the simplified version of Y. Hence, surfel *s* must next be classified against Q. However, when one of the last two simplification rules are performed (*F*∩N⇒*F* or *T*∪N⇒*T*), the left most symbol of the simplified Y expression is the constant *T* or *F* and further simplification is possible. Consequently, Q and possibly subsequent primitives could be **skipped** because they will not affect the classification of *s*.

If we were classifying a single surfel against Y, we could easily track which primitives need to be skipped, by either performing the tree simplifications discussed above or by implementing a recursive evaluation as follows:

```
proc eval(N,s) {
   if is_primitive(N)
      then return point_in_primitive(N,s)
      else if is_union(N)
         then if eval(N.left_child,s)
            then return(T)
            else return(eval(N.right_child,s))
         else if eval(N.left_child,s)
            then return(eval(N.right_child,s)) else return(F)}
```

Note that point_in_primitive(N,s) returns *True* if *s* is in N and *False* otherwise. For negative primitives, the result is complemented.

Surfels at different pixels may have different classifications with respect to the various primitives, entailing different paths though this recursion. So, the individual status temporarily stored at each pixel must suffice to indicate the next step through this path.

Note that the *eval* procedure visits the leaves of the CSG tree in left-to-right order, possibly skipping some of them. The Blist approach is based on the realization that it is sufficient to store, at each pixel, the ID of the next primitive that cannot be skipped when classifying the corresponding surfel *s*. We will refer to the primitive being rasterized as the **current** primitive and to the primitive whose ID is stored at the pixel p of *s* as the **active** primitive. Hence, from the perspective of a single pixel p, skipped primitives are never active. Parity testing against an inactive primitive will not affect the status of p while parity testing against an active primitive will replace the status of p with the ID of the next active primitive for p or by an ID that represents the final result, *in* or *out*.

The concept of tree simplification is used below to explain which primitives become active. Consider *Y*=(ABC∪D)(E'F'∪G'). The first primitive, A, is active by default. If when rasterizing A, we discover that *s* is inside A, then A can be replaced by *T* and the expression becomes (BC∪D)(E'F'∪G'). Hence, the next active primitive for *s* is B and its ID will be stored as *status* at p. If, however, *s* is not in A, then the expression becomes D(E'F'∪G') and the ID of next active primitive, D, is stored in p. B and C still perform parity testing because they may be active for other pixels, but the parity results they yield is discarded for p.

To illustrate the consecutive stages of the classification, we use the term PET(P) (Partially Evaluated Tree of primitive P) to denote the state of the simplified tree when an active primitive P becomes current. Note that PET(A) is by default the original expression, (ABC∪D)(E'F'∪G'). We are interested in PET(B) only when B is active. Since B would be skipped if A was replaced by *False*, we can assume that A was *True* (i.e. *s* was in A) and that PET(B) was produced by simplifying (*T*BC∪D)(E'F'∪G'), which yields (BC∪D)(E'F'∪G'). The PETs for all of the primitives in (ABC∪D)(E'F'∪G') are shown in Table 1.

Furthermore, note that if primitive P is active for *s* when it becomes current, then depending on whether the classification of *s* against P returns *T* or *F*, the next active primitive for *s* will either be the primitive corresponding to the next symbol in the CSG expression or one farther along the list. When it is the **next** primitive, we store in the status of p the ID of that primitive. Otherwise, we store the ID of the left-most primitive of the simplified version of PET(P), with the exception that if the tree degenerates to *T* or *F*, we store the ID assigned to *in* our *out* respectively. We will explain later how to avoid having to treat these situations as special cases. We call the resulting content of the status of p "the **match** of P" and denote it match(P). Note that there is only one match for each primitive (bold letter in Table 1). Furthermore, note that match(P) may appear in either the third column or fourth column of Table 1.

P	PET(P)	PET(P) if P=T	PET(P) if P=F
A	(ABC∪D)(E'F'∪G')	(BC∪D)(E'F'∪G')	**D**(E'F'∪G')
B	(BC∪D)(E'F'∪G')	(C∪D)(E'F'∪G')	**D**(E'F'∪G')
C	(C∪D) (E'F'∪G')	**E'**F'∪G'	D(E'F'∪G')
D	D(E'F'∪G')	E'F'∪G'	***Out***
E'	E'F'∪G'	F'∪G'	**G'**
F'	F'∪G'	***In***	G'
G'	G'	***In***	*Out*

Table 1: Column 2 shows the PET(P) of each primitive P. Column 3 shows what this PET would simplify to if s were in P. A bold symbol identifies match(P).

P	PET(P)	match(P)	flip(P)
A	(ABC∪D)(E'F'∪G')	**D**	*false*
B	(BC∪D)(E'F'∪G')	**D**	*false*
C	(C∪D)(E'F'∪G')	**E'**	*true*
D	D(E'F'∪G')	***out***	*false*
E'	E'F'∪G'	**G'**	*false*
F'	F'∪G'	***In***	*true*
G'	G'	***In***	*true*

Table 2: For each primitive P (column 1) we show its PET (column 2), its match (column 3) and its flip (column 4).

We have shown that one can associate with each primitive P of a positive CSG expression a unique ID called match(P), which may be the ID of a primitive or a special symbol for *next*, *in,* or *out*. Furthermore, we associate with each primitive P a binary flag, flip(P), indicating whether match(P) should be stored at the pixel when s is **in** P (third column in Table 1) or when s is **out** of P (4th column). Specifically, when point_in_primitive(P,s)=flip(P), we store match(P) at the pixel of *s*. Otherwise, we store *next*. The values of match(P) and flip(P) are shown in Table 2 for our example case.

3.2 CSG to Blist Conversion

The above process allows us to represent an arbitrary Boolean expression as a Blist (list of modules), associated bijectively with the consecutive variables as they appear in the Boolean expression (Fig. 8). In the module m associated with the variable P, we store the name m.name, which is P, m.match, which is match(P), and m.flip, which is flip(P).

When evaluating the expression, we start by sending the default status, *next*, to the first module. Each module m has two inputs. These inputs are the *status* received from the previous module in the list and a Boolean value, m.value, which in our case represents the classification of *s* against primitive m.name.
When status≠m.name, the module simply passes *status* to the next module. If status=m.name, the output of m is based on the following algorithm:

```
if status=m.name then
    if m.value=m.flip then status:=m.match else status:=next;
```

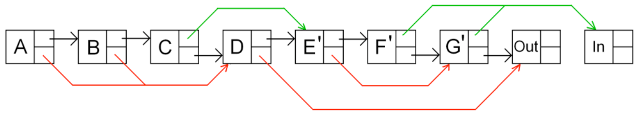

Figure 8. For each module, m, we show m.name. The arrow from the upper square points to the next active module for cases when m.value is *True*; the downward arrow for cases when m.value is *False*. When m.flip is *True*, the upward arrow points to the next module.

Note that this solution departs significantly from previously reported efforts to linearize Boolean expressions [Kohavi 1986], because the status is not a Boolean value or set of Boolean values, but the address (or ID) of a single primitive.

When applying this concept to the GPU-based rendering of CSG models, we need to distinguish between complemented (negative) primitives, the interior of which is unbounded, from the non-complemented (positive) primitives. Hence, with each module, we also store a flag m.complemented, which is set for negative primitives.

It should be clear from the above discussion that the match and flip of a primitive P could be computed through the iterative simplification of the CSG expression. Instead, we have chosen to implement them using the following recursive algorithm:

```
Q := P;
while Q ≠ Q.parent.leftChild do Q:=Q.parent;
op:=Q.parent.operator;
if op==“∩” then flip := false else flip:=true;
Q:=Q.parent;
WHILE (Q≠Q.parent.leftChild)
      OR (Q.parent.operator=op) DO Q:=Q.parent;
Q:=Q.parent.rightChild;
WHILE Q.type≠ eaf DO Q:=Q.leftChild;
match:= Q.name;
```

In the first two lines, we walk up the tree until the current node Q is the left child of some node R. The operator op of R is remembered and used to set *flip*. Note that for negative primitives, we reverse *flip*, as discussed earlier. We continue walking up until we reach, from the left, a node S with an operator that is not op. The *match* is the left-most leaf of the right-subtree of S. The matches for the primitives in our example are shown in Fig. 9.

To avoid special cases for the end-conditions, we imbed Y in (Y∪*out*)∩*in*), with *in* returning *True* and out returning *False*.

3.3 Storage reduction of status information

So far, we have implied that we will store the names of the primitives or one of the two other values (*in*, *out*) in the status of a pixel. We could do this using $\lceil \log_2(n+2) \rceil$ stencil bits, where n is the number of primitives in the CSG expression. We explain below how to further reduce this storage cost.

Our approach is based on the observation that we can rename primitives in order to **reuse** IDs. As we compute the match values for all primitives, we maintain a list of free positive integers and use the lowest one available.

When we arrive at a given primitive M with M.id, we can free M.id and allow a subsequent primitive to use it. The following greedy algorithm creates a minimized set of IDs.

```
Initialize every node to name –1.
For each node m
    Unlock(m.id )
    if m.next.id = –1 then m.next.id:=LockLowestId()
    if m.match.id = –1 then m.match.id:=LockLowestId()
```

The result, for our example, is demonstrated in Fig. 10.

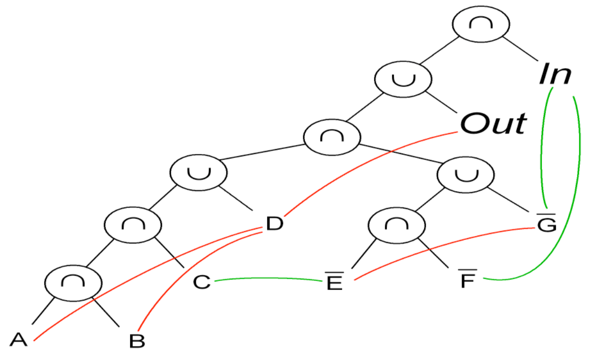

Figure 9: The green arcs show matches for primitives without flip. The red arcs show matches of flipped primitives.

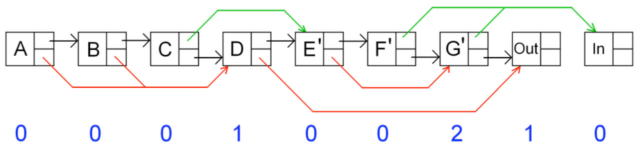

Figure 10: The corresponding IDs for each node.

Let *path* be the set of nodes traversed when walking from the root R to a leaf representing the current primitive Q. A walk between a previously visited primitive to its not yet visited *match* must reach *path* from a left child of a node in *path* and must leave *path* to a right child of a node in *path*. If two walks overlap along *path*, they have the same match. Hence, the number of ID’s still in use when Q is reached cannot exceed the number of zigzags in *path*, which is bounded by half the height h of the tree plus one. For a tree with n leaves, the worst-case scenario is illustrated on Fig. 11 (left), where the number of IDs needed is N/2+1.

However, we can rearrange the tree to make it left-heavy (Fig. 11 right). This may be done through a recursive traversal, during which, at each node, we recursively swap children when necessary such that the left sub-tree of any parent is deeper than the right-sub-tree. This rearrangement removes unnecessary zigzags.

The tree on the left of Fig. 11 may already be left-heavy if D, F, and H are not leaves but sub-trees of height 2, 4 and 6 respectively. Hence, the minimum number n of leaves in such a tree is 21. More generally, the minimum number n of leaves in a tree that requires x IDs is 3+(2+4+6+…+x), which yields N=x(x+1)+3. Contemporary graphics adapters offer 8-bit stencil buffers, but two of these are needed for other purposes. Thus, we can offer $x=2^6=64$ different IDs. Considering that *in* and *out* have their own IDs, we can guarantee support of all CSG expressions with up to 62*63+3=3909 primitives. Because contrived worst-case arrangements are rare, most trees with a significantly larger number of primitives will still work with 6 available stencil bits.

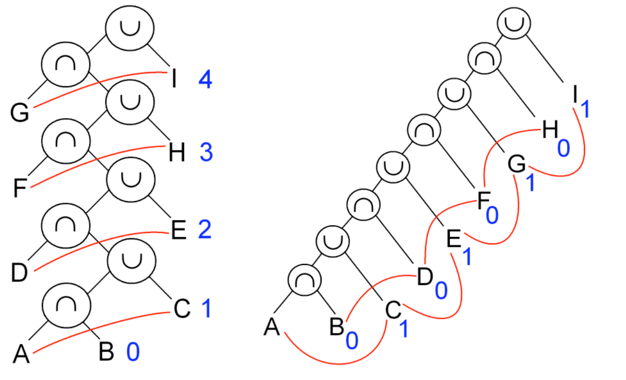

Figure 11: The original tree (left) requires 5 IDs. Rearranging it into a left-heavy tree requires only 2 IDs. Note that a list, such as the right tree, requires only 2 IDs, regardless of its length.

4 Blister

We assume that a Blist formulation has been calculated as explained in the previous section. To render the corresponding CSG object, Blister processes one depth-layer at a time obtained using a variation of the Depth Interval Buffer. Assume that the results of peeling and processing previous layers are stored in buffer $\mathbf{Z_S C_S}$, which contains both the z values in $\mathbf{Z_S}$ and the color values in $\mathbf{C_S}$. We also store a color buffer $\mathbf{C_F}$ with no associated depth buffer that holds the final image. A buffer $\mathbf{Z_B}$ stores an auxiliary depth used to reduce the total number of layers.

4.1 Layer Extraction and Composition

For the current layer, we rasterize all the primitives, perform a parity test for each, and use the Blist logic to update the *status* at each pixel. Surfels of the new layer that pass the test are merged into $\mathbf{C_F}$. This process is illustrated in Fig. 7.

To create $\mathbf{Z_B}$, consider that the current primitive Q is active for a pixel p that is not covered by the projection of Q on the screen. We make the observation that we need only visit the pixels covered by the active zone Z(Q) of Q [Rossignac and Voelcker 1988], which is the region where the shape of Q matters. Because Z(Q) is expensive to calculate, we use a superset. We select a subset R of the original primitives whose union is guaranteed to contain the entire CSG solid. We compute R by observing that A–B⊂A, A∩B⊂A, and A∩B⊂B. Hence, we traverse the original CSG tree (before it was converted to positive form) and collect lists of primitives up the tree. A "–" node returns the collection of its left child; a "∪" node returns the union of the collections of its two children; and a "∩" node returns the smallest of its children's collections. Note that we could also use s-bounds [Cameron 1992] instead of the primitives to compute such a container. Observe that no pixel that falls outside of R can contribute to our scene. Thus, we initialize our first layer of $\mathbf{Z_S C_S}$ to the front-most layer of R, and $\mathbf{Z_B}$ to the back-most layer. We then classify every pixel in $\mathbf{Z_S C_S}$ as *in* or *out* and advance on to the next peel.

For each subsequent peel, we render each primitive while discarding any pixels that are in front of $\mathbf{Z_S}$ or behind $\mathbf{Z_B}$. Since $\mathbf{Z_S}$ and $\mathbf{Z_B}$ are initialized to the furthest and closest layers of R, we only evaluate the subset of Q that is inside R. We also discard pixels for which $\mathbf{C_F}$ has already stored a completely opaque pixel, since any pixel that is behind a completely opaque pixel will not be visible. The steps for discarding these pixels are performed in a pixel shader. The standard Z-buffer test yields the front-most set of surfels, which constitute the next layer. We replace $\mathbf{Z_S}$ with the new layer and continue extracting, classifying, and merging surfels until $\mathbf{Z_S}$ falls entirely behind $\mathbf{Z_B}$. These added early-rejection steps provide significant performance gains.

4.2 Layer Classification

To evaluate the pixels in $\mathbf{Z_S C_S}$, we divide the stencil buffer into a one bit *parity* mask for parity testing, a six bit *state* mask for storing that tag needed in Blist evaluation, and a one bit *utility* mask for stencil value switching. We initialize the *state* mask to the first element in the Blist. For each primitive, we perform a parity test against $\mathbf{Z_S C_S}$, storing the result in the *parity* mask. For a pixel p testing against that primitive in the Blist, we set the state at p to P_{IN} if we are inside and to P_{OUT} if outside, assuming that P is active for that pixel. We have implemented this step using the following three operations:

1. Set *utility* to 0
2. If *state* = P and *parity* = 1, then set *utility* to 1
3. If *utility* = 1, then set *state* to P_{IN}

We then perform the process again for P_{OUT} when *parity*=0. Eventually, the state for every pixel of $\mathbf{Z_S C_S}$ will be *in* or *out*, and we merge the *in* pixels into $\mathbf{C_F}$.

4.3 On/on cases

So far, we have assumed that each surfel lies either inside a primitive or outside of it. Unfortunately, surfels of a primitive Q lie on the boundary of Q. Furthermore, they may also lie on other primitives that share portions of their boundary with Q. Rossignac and Requicha [1986] and Rossignac and Wu [1992] have solved this problem by classifying a surfel *s'* slightly behind surfel *s*. This offset technique is trivial to implement using a slight perturbation of the perspective matrix or polygon offset. When the candidate surfel *s'* is tested against the primitive that contains *s*, rasterizing the primitive at *s* will add one extra inversion to the parity checking, which provides the correct results. The drawback lies in the fact that surfels near sharp silhouette edges, surfels on walls thinner than a z-buffer resolution may be misclassified.

We have replaced this depth-offset heuristic with a new classification that produces correct results, given the quantized depth information at each pixel. In that sense, it is consistent with the definition of the boundary of the regularized version of the CSG solid [Tilove 1980, Requicha and Voelcker 1982] after quantization.

To support our new approach, we alter the parity test to invert its state if the rasterized point *p* behind or at the same depth as our surfel, *s* (Fig. 12). Toggling parity when *p* lies exactly on *s* causes the same parity inversion as testing *p* against *s'*.

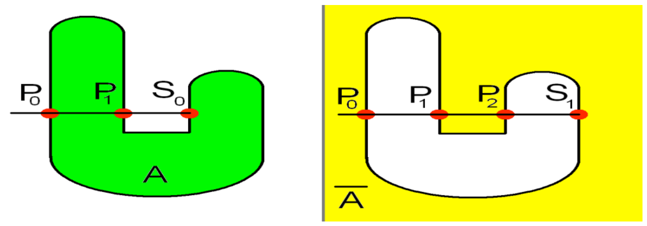

Figure 12: When rasterizing front-faces of positive primitives and back-faces of negative primitives, p_0, p_1, and p_2, toggle the parity bit when the rasterized surface coincides with the candidate surfel.

Blister correctly classifies surfels based on their depth and hence produces the correct depth map for the CSG solid. However, it does not differentiate between two surfels that project onto the same pixel and happen to have the same quantized depth, because both are tested against the same Blist expression. Consequently, when these surfels have different colors or normals, the first one encountered by Blister will be used. When each one of the two surfels lies within the **active zone** [Rossignac and Voelcker 1988] of the corresponding primitive, the color of that point is ambiguous and hence it is correct to use either one. However, when one of the surfels lies outside of the active zone of its primitive, that surfel may be rendered with the wrong color or wrong normal. Therefore, instead of classifying all the surfels against the same Blist expression of the CSG model, one may chose a slower variation of Blister that classifies surfels against the Blist of the active zone of its source primitive as suggested by Rossignac [1994]. The visible surfels of the active portion of each primitive would be merged using their depth into the final picture.

5 Results, Implementation, and Extensions

We demonstrate performance on Widget, Complex, Gear, and Bunny (CSG models X_1, X_2, X_3, and X_4 in Fig. 0 respectively) and on Bunny2 obtained from Bunny through subdivision. Table 3 shows: the number n of primitives, the number p of products after pruning, the number k of peels (averaged for various directions), the number k_f of peels after pruning unnecessary pixels, the time T needed to render the model when all k peels are rendered, and the time T_f when we only use k_f peels. Timings are in milliseconds on the NVIDIA BFG GeForce 6800 OC card.

	n	p	k	k_f	T	T_f	T_f/Nk_f
Widget	4	1	4.00	3.85	17	18	1.17
Bunny	4	3	9.13	6.81	405	325	11.9
Bunny2	4	3	9.09	6.77	1.5s	1.2s	44.3
Complex	20	48	14.1	9.17	139	93	0.51
Gear	48	31104	36.4	25.6	636	460	0.37

Table 3: Results with all times in milliseconds.

As the number of primitives in an object becomes large, the total rendering time becomes proportional to Nk_f, which is far better than the exponential time needed for algorithms based on the disjunctive form of the CSG expressions. The Bunny and Bunny2 models have 574,848 and 2,299,392 triangles respectively. For these two models, the bottleneck is vertex processing, and we verify that increasing the tessellation proportionally increases rendering time. The difference between T and T_f shows the effectiveness of removing unnecessary pixels from $\mathbf{Z_S C_S}$. In general, about half of the time is spent peeling, whereas the other half of the time in Blister is spent in the evaluation of the candidate surfels.

As noted by Everitt [2002], to render a transparent model, layers must be shaded in depth order. Blister extracts surfels from the CSG in depth order, and thus is trivially extended to support **transparency** (as illustrated in the last image of Fig. 0).

Rendering **shadows** directly from CSG required modifying the original depth peeling algorithm the correct distances are stored in the z-buffer. Instead of clearing the z-buffer before every peel, we render a quad over the screen and clear the z-buffer only at pixels which do not have a completely opaque surfel. Consequently, the Z-buffer stores the correct depth for each pixel and hence may be trivially extended to support techniques such as shadow mapping.

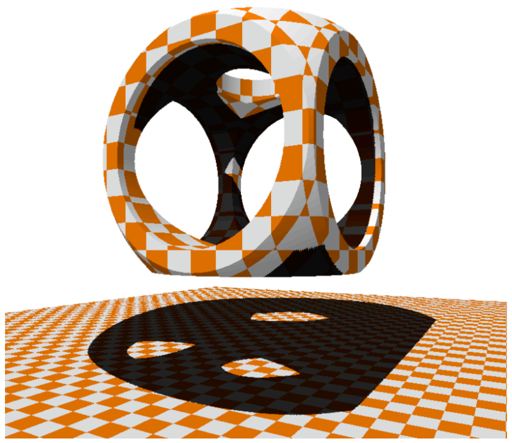

Figure 13: A CSG object with shadow mapping.

In the worst case, when k, the depth complexity of the arrangement of primitives, is a linear function of n, Blister has $O(n^2)$ **time complexity**. Note however that such extreme situations are rare. For example, the depth complexity of a city with n buildings would be a constant for aerial views and would grow as $\sqrt{n}$ for horizontal views. Similarly, the depth complexity of a CSG model made by subtracting n randomly spaced small balls from a block would grow as the cubic root of n. In most cases, k is a sub-linear function of n.

6 Conclusion

We have introduced the new Blist representation for CSG expressions. We have proposed the Blister algorithm, which allows complex CSG expressions, represented in Blist form, to be rendered on inexpensive graphics hardware in real-time. Blister may be used to support transparency and shadow mapping directly from CSG.

7 Acknowledgements

We thank Spencer Reynolds from Georgia Tech for editing our video, Ying Chen and Andy Scukanec from Georgia Tech for providing implementations of the CSG-to-Blist conversion, an anonymous reviewer for suggesting that Blister could be used for shadow mapping, and Mark Kilgard from NVIDIA for pointing out that the *status* update (Section 4.2) can be implemented with only 2 stencil passes and without a *utility* mask. His solution is to use OpenGL stencil operations to flip the bits of *status* identified by the XOR between the ID of P and either P_{IN} at pixels where *state*=P and *parity*=1, or P_{OUT} at pixels where *state*=P and *parity*=0. This change would improve performance and increase the complexity of acceptable CSG expressions, since 7 stencil bits would be available for storing the *status*.

References

ADAMS, B., AND DUTRÉ, P. 2003. Interactive Boolean operations on surfel-bounded solids, *ACM Transactions on Graphics*, 22, 3, 651-656.

BANERJEE, R., GOEL, V., MUKHERJEE, A. 1993. Efficient parallel evaluation of CSG tree using fixed number of processors. *ACM Symposium on Solid Modeling and Applications*, 137-146.

BARBER C., DOBKIN, D., HUHDANPAA, H., 1993. The Quickhull algorithm for convex hulls, *ACM Transactions on Mathematical Software*, 22, 4, 469-483.

CAMERON, S., YAP, C. 1992. Refinement methods for geometric bounds in constructive solid geometry, *ACM Transactions on Graphics*, 11, 1 12-39.

CHEN, H., AND FANG, S. 1999. A volumetric approach to interactive CSG modeling and rendering. *ACM Symposium on Solid Modeling and Applications*, 318-319.

ELLIS J., KEDEM G., LYERLY G. T., THIELMAN D., MARISA R., MENON J. 1991. The Ray Casting Engine and ray representations. *ACM Symposium on Solid Modeling Foundations and Applications*, 255-268.

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 Vurtual Reality und Visualisierung Forschungs-GmbH.*

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

FANG, S., AND LIAO, D. 2000. Fast CSG voxelization by frame buffer pixel mapping. *IEEE Symposium on Volume Visualization*, 43-48.

FUCHS, H., AND POULTON, J. 1981. Pixel-planes: a VLSI-oriented design for 3-D raster graphics. *Canadian Man-Computer Communications Conference*. 343-347.

GOLDFEATHER, J., HULTQUIST, J. P. M., AND FUCHS, H. 1986. Fast constructive solid geometry display in the pixel-powers graphics system. *Annual Conference on Computer Graphics and Interactive Techniques,* 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, 20-28.

GOODRICH, M. 1990. Applying parallel processing techniques to classification problems in constructive solid geometry. *ACM-SIAM Symposium on Discrete Algorithms*, 118-128.

GOODRICH, M. 1998. An improved ray shooting method for constructive solid geometry models via tree contraction. *International Journal of Computational Geometry and Applications*, 8, 1, 1-24.

GUHA, S., KRISHNAN, S., MUNAGALA, K., VENKATASUBRAMANIAN, S. 2003. Application of the two-sided depth test to CSG Rendering. *Symposium on Interactive 3d graphics*, 177-180.

HOFFMANN, C. 1989. *Geometric and Solid Modeling: An Introduction,* Morgan Kaufmann.

JANSEN, E. 1986. A Pixel-Parallel Hidden Surface Algorithm for Constructive Solid Geometry. *Eurographics,* 29-40.

KELLEY, M., GOULD, K., PEASE, B., WINNER, S., YEN, A. 1994. Hardware accelerated rendering of CSG and transparency. *Conference on Computer Graphics and Interactive Techniques*, 177-184.

KEYSER, J., BULVER, T., FOSKEY, M., KRISHNAN, S., MANOCHA, D. 2002. Esolid – A system for exact boundary evaluation. *ACM Symposium on Solid Modeling and Applications*, 23-34.

KOHAVI, Z. 1986. *Switching and Finite Automata Theory, 2nd Edition*. McGraw Hill College. R. Hamming and E. Feigenbaum.

LIAO, D., AND FANG, S. 2002. Fast volumetric CSG modeling using standard graphics system. *ACM Symposium on Solid Modeling and Applications*, 204-211.

MAMMAN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9, 4, 43-55.

RAPPOPORT A. AND SPITZ, S. 1997. Interactive Boolean Operations for Conceptual Design of 3-D Solids. *ACM SIGGRAPH Conference on Computer Graphics*, 269-278.

REQUICHA, A., AND VOELCKER, H. 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 75, 1, 30-44.

ROSSIGNAC, J., AND REQUICHA, A. 1986. Depth-buffering display techniques for constructive solid geometry, *IEEE Computer Graphics and Applications*, 6, 9, 26-39.

ROSSIGNAC, J., VOELCKER, H. 1988. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms, *ACM Transactions on Graphics*, 8, 1, 51-87.

ROSSIGNAC, J., AND WU, J. 1992. Correct shading of regularized CSG solids using a depth-interval buffer, *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, Eurographics Seminars, 117-138.

ROSSIGNAC, J., MEGAHED, A., SCHNEIDER, B.O. 1992. Interactive Inspection of Solids: Cross-Sections and Interferences, *ACM Computer Graphics, Vol. 26, No. 2, pp. 353-360, July (Proc. SIGGRAPH).*

ROSSIGNAC, J. 1994. Through the cracks of the solid modeling milestone, *From Object Modeling to Advanced Visual Communication,* S. Coquillart, W. Strasser, P. Stucki, Ed., Springer-Verlag, 1-75.

ROSSIGNAC, J. 1996. CSG formulations for identifying and for trimming faces of CSG models. In *CSG'96: Set-theoretic solid modeling techniques and applications,* Information Geometers, Ed. John Woodwark. 1-14.

ROSSIGNAC, J. 1999. BLIST: A Boolean list formulation of CSG trees, Technical Report GIT-GVU-99-04 available from the GVU Center at Georgia Tech. *http://www.cc.gatech.edu/gvu/reports/1999/*

STEWART, N., LEACH, G., AND JOHN, S. 1998. An improved z-buffer CSG rendering algorithm. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, ACM Press, New York. 25-30.

STEWART, N., LEACH, G., AND JOHN, S. 2000. A CSG rendering algorithm for convex objects. *International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media – WSCG 2000*, 2, 369-372.

STEWART, N., LEACH, G., AND JOHN, S. 2002. Linear-time CSG rendering of intersected convex objects. *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision – WSCG 2002*, 2, 437-444.

STEWART, N., LEACH, G., AND JOHN, S. 2003. Improved CSG rendering using overlap graph subtraction sequences.*International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, ACM Press, New York. 47-53.

TAWFIK, M. 1991. An efficient algorithm for CSG to Brep conversion. *ACM Symposium on Solid Modeling Foundations and Applications*, 99-108.

TILOVE, R.B. 1984. A Null-Object Detection Algorithm for Constructive Solid Geometry, *Communications of the ACM*, 27, 7, 684-694.

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