CST: Constructive Solid Trimming for rendering BReps and CSG

John Hable and Jarek Rossignac

Abstract— To eliminate the need to evaluate the intersection curves in explicit representations of surface cutouts or of trimmed faces in BReps of CSG solids, we advocate using Constructive Solid Trimming (CST). A CST face is the intersection of a surface with a Blist representation of a trimming CSG volume. We propose a new, GPU-based, CSG rendering algorithm, which trims the boundary of each primitive using a Blist of its Active Zone. This approach is faster than the previously reported Blister approach, eliminates occasional speckles of wrongly colored pixels, and provides additional capabilities: painting on surfaces, rendering semitransparent CSG models, and highlighting selected features in the BReps of CSG models.

Index Terms— I.3.3 GPU support for CSG rendering, 3.5.a&b CSG expressions for trimmed faces, J.6 CAD model visualization

1 INTRODUCTION

MANY design, visualization, analysis, and entertainment applications manipulate models of solids. A common approach is to represent a solid by a triangle mesh that approximates its boundary.

A mesh M that is **bounded** (i.e. finite) and **watertight** (in which each edge has an even number of incident triangles) divides its complement into two half-spaces: the **interior** i(M) and the **exterior** e(M) of M. i(M) is the set of points from which rays that avoid the edges and vertices of M stab an odd number of triangles of M.

Let M.v, M.e, and M.t be respectively the set of vertices, edges, and triangles of M. In our terminology, an edge does not include its bounding vertices and a triangle does not include its bounding edges and vertices. We say that M is **clean** when the set of all triangles, edges, and vertices of M are exclusive (i.e. pairwise disjoint).

The **regularization** r(V) of a set is the closure of its interior. The **boundary** of a regularized solid separates its interior from its exterior. For simplicity, we say that a set V is a **solid** when V=r(V). When M is bounded, watertight, and clean, then the union V=M∪i(M) is a solid and M is its boundary.

A **CSG** (Constructive Solid Geometry) representation of V defines it as a **regularized** Boolean expression [1] that combines primitive solids through union (+), intersection (omitted), and difference (–) operators. We denote the complement of a primitive A as !A. In what follows, we discuss **CSG** representations that define solids as regularizations of Boolean combinations of solid primitives, each defined by a clean mesh. For example, the solid in Fig. 1.b is defined by the Boolean combination of 20 different primitives.

We propose a new GPU-based approach for the realtime rendering of CSG models and of trimmed surfaces (the intersection or differences between a surface represented by a triangle mesh, which needs not be watertight or clean, and a CSG model).

Complex CSG models can be rendered in realtime on commodity graphics adapters by exploiting the observation that the boundary of a CSG solid is a subset of the boundaries of its primitives. Hence, many approaches rasterize the surfaces that bound the primitives, identify points that are ON the solid's boundary, and select the front-most boundary-point at each pixel. The differences between approaches lie in the order in which the primitives are rasterized and in the manner in which candidate points are tested to establish whether they are boundary-points or not. For instance, the Blister approach [2], generates candidate points by layers using hardware-supported front-to-back peeling which stores the candidate points of the current layer in a depth-buffer. It tests all the candidates of the current layer against the entire CSG model by scan-converting each primitive; by keeping track of the parity of the number of triangles of the primitive that occlude each candidate point; and by merging the results using only a few book-keeping stencil bits per pixel.

Unfortunately, a candidate point generated by scan-converting a primitive A may happen to lie on another primitive B. This singular situation may reflect the designer's intention of using primitives with overlapping faces or it may be an unfortunate coincidence produced by performing z-buffer tests on quantized depth values. (Even though two candidate points that project on the same pixel may have different depths, their quantized versions may be identical.) Consequently, candidate surface points on A that lie close to the boundary of B are often misclassified and displayed using the surface properties of B, producing wrongly colored **speckles** in the image (See Fig. 2). Note that even though the color at speckles may be wrong, their quantized depth is correct. The CST technique proposed here eliminates such speckles without reducing performance.

- *John.W. Hable is with the Worldwide Visualization Group, Electronic Arts, Vancouver, BC V6C 3R8, Canada. Formerly with Georgia Institute of Technology.*
- *Dr. Jarek R. Rossignac is member of GVU and Professor of Computer Science in the Interactive Computing Department, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: jarek@cc.gatech.edu.*

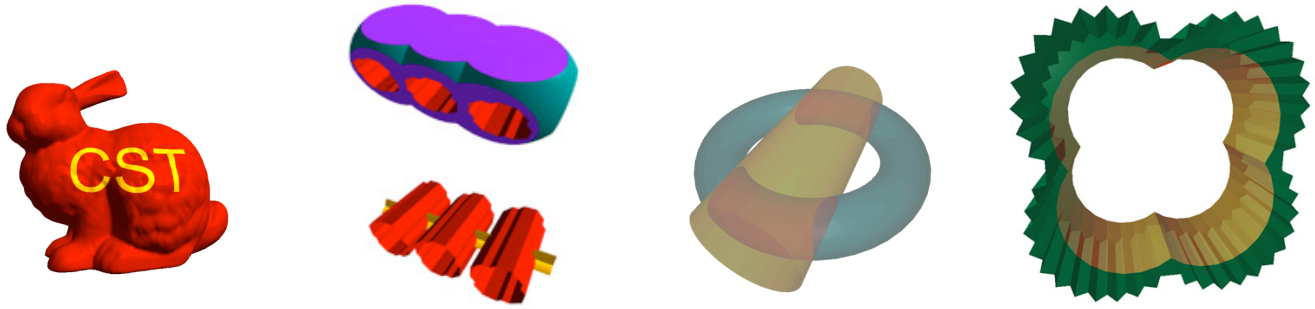


Fig. 1. Examples of CST applications include (from left to right): (a) Using extruded solids to carve or paint on the "Bunny"; (b) revealing the trimmed faces of selected primitives (features) in the BRep of the "Complex" CSG model; (c) highighting (in red) the interference between two semitransparent solids in a small assembly; and (d) correctly rendering a semitransparent CSG model with numerous overlapping primitive boundaries (such as the "Gear" shown here). These models were rendered in realtime on the GPU using our peel&trim process.

In fact, most other previously proposed techniques [3], [2] do not really test whether a candidate point lies on the boundary of the CSG solid. Instead, they only differentiate between candidate points that are OUT (i.e. in the exterior of the CSG solid) from those that are IN/ON (i.e. either in the interior or on the boundary of the solid). This distinction is sufficient for rendering, since, assuming that the viewpoint is outside of the solid, the front-most IN/ON point selected by the depth-test at each pixel is necessarily ON the solid [4].

Unfortunately, these approaches are not able to discriminate whether IN/ON points that are hidden by a front-most IN/ON point lie on the boundary of the CSG solid or in its interior. This situation is illustrated in Fig. 3, which shows a 2D slice through the model.

With such approaches, the union of all IN/ON points of a CSG solid V cannot be used as a rasterized representation of V when one wishes to use the parity test for classifying candidate points of another mesh S against V.

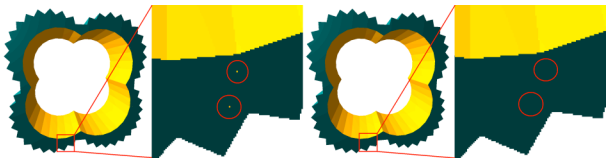


Fig. 2. Blister with speckles, and CST which removes them.

This drawback leads to several limitations, all of which are removed by the CST approach proposed here.

Limitation 1: The techniques mentioned above are not suitable for rendering CSG models as if their boundaries were **semi-transparent**, because in most cases they render extraneous portions of the boundaries of primitives that are IN/ON, but not necessarily ON S. Some approaches [5], [6], [7], [8], [9], [10] convert a CSG expression into a disjunctive form [11] which is the union of intersections (products) of primitives. These approaches are able to distinguish the ON points of each product from the IN points. Unfortunately ON points of a product may be IN points of the union of products. Rendering them as semi-transparent produces incorrect and confusing images.

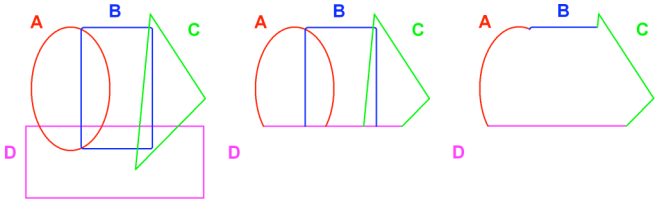


Fig. 3. Four primitives (left). The boundary of the solid V=(A+B+C)–D (right). IN/ON points (center).

Limitation 2: The rendering system is not capable of correctly coloring the **contribution of a primitive** A to the boundary of the solid V. Such a facility is vital during the design phase when the user selects and manipulates one or more primitives at a time and needs to see clearly which portion of the solid's boundary are affected by changes to the selected primitives.

Limitation 3: The inability to distinguish the ON points from the other IN points of CSG models prevents the use of CSG models as **trimming volumes** for surfaces represented by meshes that are not water-tight. This facility is important for a broad set of geometric modeling and artistic design applications, in which Boolean combinations of solids are used to trim faces or to paint logos or patterns on faces (Fig. 1.a).

The removal of the above limitations by the CST approach introduced here hinges on a single capability: trim a surface S against a solid or CSG model V returning, depending on the application, the portion of S in V, out of V, or on V. We call this function CST(S,V). This trimming capability is based on two operators implemented efficiently on commodity GPUs:

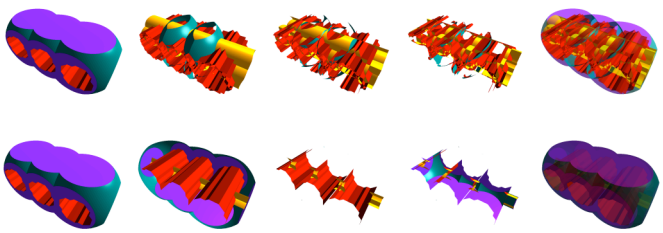


Fig. 4. Four consecutive layers generated by Blister (top) and by our CST approach (bottom) are combined (right) to produce a semi-transparent image. Note that the CST approach produces the correct image (bottom-right) while Blister does not (top-right).

Peel(S) pushes depth values at unlocked pixels to candidate points on the next depth-layer of a mesh S during the rasterization of S.

Trim(V) classifies candidate points at unlocked pixels with respect to a Blist form of the CSG solid V and locks the pixels at candidate points that lie in V (or if preferred, out of V).

Alternating these operators until all pixels are locked or all layers of S are processed renders a **Constructive Solid Trim** (CST) of surface S. When S is opaque (not semi-transparent), CST(S,V) is the front-most set of pixels of S that falls inside the solid defined by V.

We advocate using such CSTs as an option for **rendering CSG** models quickly and reliably. We realize that in many applications, it may be preferable to pre-compute and triangulate the boundaries (**BReps**) of CSG models and then to use them for rendering. However, this pre-computation is often numerically delicate and too slow for realtime feedback during shape design or animation of parametric CSG models.

For instance, boundaries of CSG models with polyhedra primitives may be pre-computed reliably [12], triangulated, and used for realtime rendering. However, such a boundary evaluation is too slow (even for polyhedral models) to be invoked at each frame during interactive editing or when the tessellations of the primitive boundaries must be refined through subdivision during camera motions. Hence, numerous direct CSG rendering algorithms have been proposed that eliminate this boundary evaluation cost. CST improves on them providing a fast, accurate, and scalable solution.

Furthermore, our CST approach offers an **alternative to** the trimmed-surface representation of **BReps**. Using CST for rendering BReps makes it possible to render trimmed NURBS [13], [14] and subdivision surfaces without having to parameterize them, compute intersection curves, and convert these curves into trimming curves in the parametric domain. CST also eliminates the need to stitch the cracks between abutting trimmed faces [15], [16] and of tracking the intersection curves in animated models [17]. Note that this option is only possible when a CSG expression of a trimming volume may be derived for each face, which is the case when the shape is created through Boolean operations, but may not be the case when the trimming curves are generated for example by filleting procedures.

Hence, we propose two techniques, both based on the peel&trim approach mentioned above. One renders CSG models, the other one renders trimmed faces of BReps, where the trimming volume is a solid defined by a CSG expression.

Our CST approach to CSG rendering is based on a new algorithm, which **peels each primitive P** of a CSG model V and classifies it against the **Blist** of the active zone Z of P. The **active zone** Z is the volume where changes of P affect V [18]. Its CSG formulation may be derived algorithmically from the CSG expression of V. The Blist form of a CSG expression [19] reduces the storage per pixel needed for combining the results of point/primitive tests when classifying a point against a CSG solid.

Using this CST approach for **CSG rendering** offers several advantages over prior approaches: (1) It **eliminates artifacts** (such as speckles). (2) It makes it possible to **highlight** the **contribution** of selected primitives. (3) It permits correct rendering of images of **semi-transparent** CSG models.

Our CST approach to the rendering of **trimmed faces** of a BRep peels each untrimmed surface S and classifies each layer of candidate points on S against the Blist representation of the CSG model of the trimming volume V.

To support CST rendering, we have developed several results reported as new **contributions** in this paper, which is organized as follows. Section 2 reviews the background and prior art for peeling, trimming, and CSG rendering. Section 3 reviews the Blist formulation, its construction algorithm, and its use for classifying candidate points against a CSG expression. Note that without the Blist approach, a large number of stencil bits per pixel could be required to combine the classification results of the corresponding candidate pixel against the CSG primitives. The Blist approach in [2] reduces that number so that candidate points may be classified against any CSG expression of 3909 primitives or less using only 7 bits per pixel. Section 4 discusses a novel, robust method for trimming a surface by a Boolean combination of solids using CST. Section 5 shows how to use CSTs for CSG rendering. Section 6 details applications of CSTs including transparency, depth peeling, and painting on faces. The implementation and speed optimizations are discussed in Section 7. Section 8 reports results and timing statistics.

2 Background and Prior Art

First, we discuss **peeling**. The ray from the viewpoint through a pixel may stab a surface S more than once. Since the front-most intersection point may be trimmed away by V, we must be able to generate the other intersection candidates (or surfels). This is accomplished by rasterizing the surface several times. Two approaches have been proposed for generating all candidate points on S. (1) Goldfeather et al. [5] used stencil bits to **count** how many times a pixel has been covered during the rasterization of a given primitive and to lock the surfel produced during the i^{th} hit of the i^{th} raster pass. (Fig. 5 left). This approach was implemented on Pixel Planes [20] and later on commodity graphics hardware [7]. (2) The Trickle algorithm [21] uses a **Depth-Interval Buffer** (DIB) [22] to traverse the layers of a product of primitives in depth-order (Fig. 5 right). The depth of the previous layer is stored in z-buffer *Front*. Z-buffer *Back* is initialized to infinity. While rasterizing S, when we find a surfel whose *depth* falls between *Front* and *Back* values of the current pixel, we replace *Back* by *depth*. The process computes the front-most layer behind *Front* and stores it in *Back*. Main memory was initially used for *Front* and *Back* [22]. Everitt [23] implemented peeling in hardware following Mamman's design [24], which uses texture memory. Our CST algorithm uses texture memory and DIB peeling to generate candidates on the consecutive layers of S. Depth-peeling custom hardware for rendering transparent CSG models has been proposed by Kelley et al. [25].

Now, we discuss how to **classify** these candidates surfels against a **single primitive** P. Several approaches have been proposed. Du and Qin [26] have used solids to trim physically defined surfaces. Schmitt at al. [27] define heterogeneous objects using Booleans between curves, surfaces, and solids. Several software approaches are possible for classifying a **fixed** sampling of candidate points on S. When P is represented as a conjunction of algebraic ine-

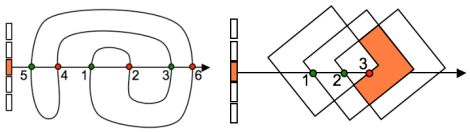


Fig. 5. Traversal-order (left) and depth-order (right) peeling.

qualities [4] candidates are rejected if they fail to satisfy any one of inequalities of the conjunction. When a voxel model or octree approximation of P provides sufficient accuracy, it may be used as a look-up table to classify candidates on S [28] or candidates inside the volume bounded by S [29]. However, when the model or the desired resolution is **changing** at each frame, it is more efficient to re-compute the candidates at each frame through rasterization and classify them using **hardware**. This is the approach that we follow.

The candidates (surfels) of the current layer are stored in the color and z-buffer of unlocked pixels. To classify them in parallel against a solid primitive P, we rasterize the boundary of P while toggling a parity bit, *parity*, at each unlocked pixel each time a rasterized portion of P lies **behind** the candidate surfel. If the final value of *parity* is true, the corresponding candidate is IN P. In fact, this approach tests whether the points immediately behind the candidate

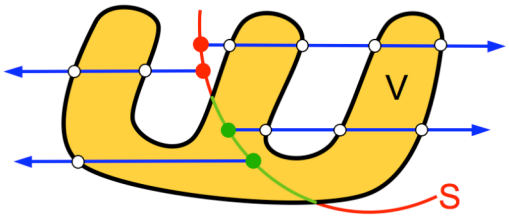


Fig. 6. Parity-based classification. Toggle parity when the rasterized primitive is on the desired side of the candidate point (red/green dots).

lies in P. Note that we could also toggle *parity* when we find P **in front** of the candidate to test whether the point immediately in front lies in P (see Fig. 6).

To classify candidates against a **CSG model** V, one can rasterize each primitive P of V, store for each candidate the corresponding *parity* results in different stencil bits, and combine these results using parallel bitwise logical operations on stencil bits [30]. Unfortunately, contemporary graphic adapters offer only 8 stencil bits per pixel. Hence, to support complex CSG expressions, most previously proposed CSG rendering approaches expand the CSG into a sum-of-products (**disjunctive form**) [21], [6] and use two stencil bits to track the status of each candidate. Several techniques were proposed to accelerate the rendering of products [21], [31], [8], [10] or other variations of disjunctive forms [9], [3]. Unfortunately, the number of products in a disjunctive form may grow exponentially with the number of primitives. Generating products may be avoided by using custom hardware to merge ray/primitive classifications [32] or, as done by Blister [2] by using a **Blist** representation [33] of the CSG expression of V. **Blister** peels the union of the boundaries of all primitives of V as if it were a single surface S and classifies the candidates on each layer against the Blist of V.

In contrast, as mentioned earlier, to improve performance, to eliminate occasional color errors, and to support a broader set of applications, the CSG rendering algorithm introduced here trims the boundary of each primitive P against a Blist of its **active zone**.

To simplify explanations, throughout the paper we assume that all CSG expressions have been converted into their **positive form** (Fig. 4), obtained by replacing each difference operator (L–R) by the intersection (L(!R)) with the complements, !R, of its right operand R and by propagating the complements to the leaves using de Morgan laws. Leaves that are complemented in this positive form (as D in Fig. 7) are said to be **negative**. The other ones are said to be **positive**.

3 Computing and Using Blists

The **Blist** form [33] of a Boolean expression is a particular case of the Reduced Function Graph (RFG) [34] and of the Ordered Binary Decision Diagram (OBDD) [35] studied for **logic synthesis**. These are Acyclic Binary Decision Graphs

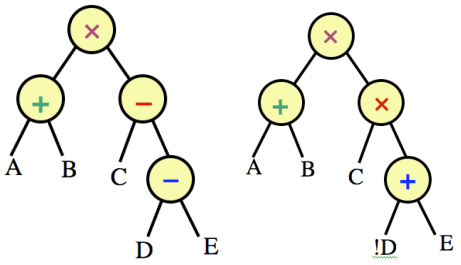


Fig. 7. CSG tree for (A+B)(C–(D–E)) and its positive form.

[36], which may be constructed through Shannon's Expansion [37]. The size (number of nodes) of RFGs may be **exponential** in the number n of primitives and depends on their order [38]. Minimizing it is NP-hard. In contrast, Blist expressions have exactly n nodes and have linear construction and optimization costs, because they treat each leaf of the tree as a different primitive. Although this may not be acceptable for logic synthesis, it is appropriate for CSG rendering. Indeed, if a primitive appears several times in a CSG expression, each instance usually has a different position, and hence must be processed as a **different primitive** during rendering.

We present below a simple algorithm for **extracting** the **Blist** of a CSG tree. Assume that the tree is stored as an **array** o of chars, where o[m] is '+' when m is a union, '×' when m is an intersection, or a letter identifying a primitive. We make two passes of linear-cost. During the **first pass** (recursive call to *lml* shown in blue below), each node *m* retrieves the name of the left-most leaf of its right child. If m represents an intersection operation, this name is stored in the field f[m], otherwise it is stored in the field t[m]. The results are marked in blue in Fig. 8.

```
char lml (int m) {    // get left-most leaf of right child
 char leftMost;
 if ((o[m]!='×')&&(o[m]!='+')) {leftMost=o[m]; }
 else {if (o[m]=='+') {f[m]=lml(r[m]); } else {t[m]=lml(r[m]); };
    leftMost=lml(l[m]); };
 return(leftMost); }
```

During the **second pass** (recursive call to *mb*, shown in red below), we push down the values of the *pt* and *pf* parameters, which are initialized to true and false (respectively) and replaced by the corresponding blue values of a node *m* when going to the left child of *m*. The resulting assignments of the t and f values (stored as t[m] and f[m] entries) are shown in red in Fig. 8.

```
void mb (int m, char pt, char pf) {   // push down blue values
 if ((o[m]!='*')&&(o[m]!='+')) {          // if leaf
  xblistName[pc]=o[m]; blistIfTrue[pc]=pt; blistIfFalse[pc]=pf;
  t[m]=pt; f[m]=pf; pc++; }
 else {if (o[m]=='+') {mb(l[m],pt,f[m]); } else{mb(l[m],t[m],pf);};
 mb(r[m],pt,pf); }; }
```

Executing our algorithm on the tree in Fig. 8 yields the Blist in Fig. 9, where each primitive P, corresponding to index m in the array representation, has 3 **labels**: (1) its ID P.n=o[m], (2) the ID P.t=t[m] of the next primitive against which a candidate that is classified as in P should be tested (shown in Fig. 9 as a link from the top right corner of the corresponding triangle), and (3) the ID P.f=f[m] of the next primitive against which a candidate that was classified out of P should be tested (shown in Fig. 9 as a link from the bottom right corner of the corresponding triangle).

For example, when P is A in Fig. 6, P.n=A, P.t=t, P.f=B. This **circuit** represents the different paths that one may take to classify a candidate X against the Blist, depending on its classifications against the primitives. For instance, if X∉A, X∈B, X∉C, X∈D, we would leave A by the bottom link to B, leave B by the top link directly to D, skipping C, and leave the whole circuit by the top link of D. Note that the special labels, t and f, of the exit links stand for true and false, and indicate the ultimate classification of X.

Since we will be storing these labels in the stencil bits of the corresponding pixels, we wish to encode them as **short** bit strings. First, as in Hable and Rossignac [2], we obtain positive **integer encodings** of the labels by traversing the Blist left to right. As we do so, we keep track of the active and free integer labels. We allocate 0 to the first primitive. When a primitive is referenced for the first time, we give it the smallest free integer label.

If one stencil bit is used for *parity*, we may represent 2^7=128 different labels using the remaining 7 bits. Since two

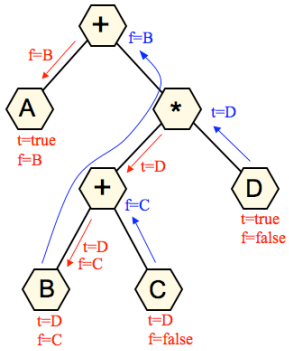


Fig. 8. Results of Iml (in blue) and of mb (in red).

labels are reserved for true and false and one for the *locked* mask, we would be able to accommodate all CSG expressions with up to 125 primitives. To accommodate **more complex CSG expressions** we **reuse labels**. For example, consider the Blist expression for (A+B)(C–(D!–E)) shown in Fig. 10. Even if we omit the label for the first primitive A, we need 5 labels. We reduce the number of labels needed (Fig. 11) by freeing and reusing the label of a primitive when it is reached by the label-to-integer conversion discussed above.

Since the intersection and union operators are commutative, one may swap (pivot) their left and right arguments to produce equivalent Blists. This flexibility may be exploited to further reduce the number of labels needed. A **pivoting strategy** that makes the tree **left-heavy** has been proposed by Hable and Rossignac [2]. It guarantees that CSG trees with 3909 leaves can be supported in Blister.

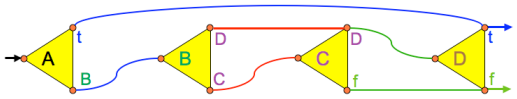


Fig. 9. Blist of A+(B+C)D with P.n, P.t, and P.f labels.

A further **optimization** [39] guarantees support of CSG trees with up to 10^{38} primitives on 7 stencil bits. For example, (A+((B+((C+((D+E)+F))+G))+H))+(I+(J+(K+L)M)N), (Fig. 10) yields a Blist form which requires 4 labels and hence 2 stencil bits. The optimization pivots the tree, producing (K+L)+JM+IN+(A+(H+(B+(G+(C+(F+(D+E)))))))), for which the Blist form requires only 3 labels.

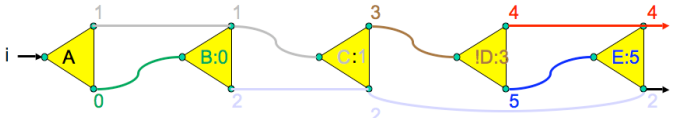


Fig. 10. Integer Blist labels for (A+B)(C–(!D–E)).

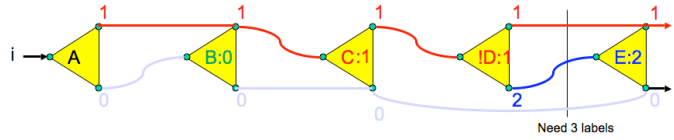


Fig. 11. Reusing labels for (A+B)(C–(!D–E)).

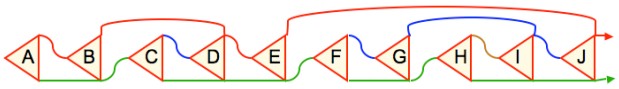


Fig. 12. The Blist of (AB+CD)E+(FG+HI)J requires 4 labels.

4 TRIMMING USING CST

In this section, we explain the implementation of our peel&trim algorithm for rendering the CST of a surface S trimmed against a CSG volume V onto a buffer called *FinalColorDepth*. The overall CST(S,V) algorithm is as follows:

Initialize;
Repeat {***Peel***(S); ***Trim***(V); ***Push***()} until ***Done***;
Render(S);

In ***Initialize***, we initialize two buffers. The first buffer, *CurrDepthStencil*, is an interleaved buffer in texture memory that associates a 24-bit depth, *Back*, and an 8-bit stencil [*next*,*parity*] with each pixel, where *next* is a 7-bit Blist label

identifying the primitive P against which the candidate is to be classified and where *parity* is toggled to establish the classification of the candidate against P. *Back* is initialized to infinity, *next* to 0 and *parity* to 0, indicating that the pixel's candidate should be classified against the first primitive of the Blist. The second buffer, *PrevDepth*, also stored in texture memory, stores a depth *Front* for each pixel. It is initialized to 0.

Peel(S) rasterizes S (or only its front faces if desired) to advance the depth of the unlocked pixels to the next layer. Let *Z denote the depth of a fragment of* S at the current pixel. We perform the update *Back*=Z, when ([*next*,*parity*]==0) && (*Front*<Z<*Back*). At the end of a *Peel*, the depth of the candidate points is stored in *Back*.

Trim(V) classifies the candidate point X of each unlocked pixel against the Blist of V. To classify a candidate point X whose depth is stored in *Back* at an unlocked pixel p, we rasterize the primitives of the Blist of V one by one. The *parity* stencil bit is toggled during the rasterization of the current primitive P to keep track of the parity of the number of layers of the boundary of P behind X.

The trimming algorithm is:
parity=false; // initialized to out
Rasterize P (toggling *parity*);
if (*next*==P.n) {if (*parity*) {*next*=P.t} else {*next*=P.f}};

For example, assume that X∉A, X∈B, X∉C, and X∈D for the Blist in Fig. 9. This algorithm will perform as follows. Initially, P is A. We reset *parity* and rasterize A. Since *next*==P.n and *parity*==false, we set *next*=P.f (label of B), to indicate that X needs to be trimmed by B. Now P=B. We reset *parity* and rasterize B. Since *next*==P.n and *parity*==true, we set *next*=P.t (label of D), to indicate that we can skip C, but need to trim X using D. Now P=C. We still reset *parity* and rasterize C, because other pixels may have *next* set to the label of C. However, the content of p will not be affected since the predicate (*next*≠P.n). Now P=D. We reset *parity* and rasterize D. Since *next*==P.n and *parity*==true, we set *next*=P.t, to indicate that X is in V.

Upon completion of *Trim*, the [*next*,*parity*] stencil of each pixel is either 0x00 (X∈V: the pixel is locked) or 0xFF (X∉V: the pixel is unlocked and will be pushed to the next layer).

Push(S) copies *Back* to *Front* at all pixels. For all pixels set to [V.t,0], the pixel's stencil state is set to locked. For all pixels set to [V.f,0], the pixel's stencil state is cleared to 0 and its depth is set to 0.

Done stops the peel&trim loop when the **occlusion query** initiated by ***Peel*** indicates that no pixel was updated during the last pass. If S is known to be the boundary of a convex set, we stop after one pass when trimming front-faces, or after two passes otherwise. Note that ***Done*** can be tested either before or after ***Trim***(V).

Render(S) merges the trimmed portion of S with the rest of the scene (which typically includes previously trimmed surfaces). When the Repeat loop stops, *Front* holds the depth-values at the retained points of S. Then, we set the rendering target to *FinalColorDepth*. S is rasterized with full **color** information updating only pixels whose depth matches the value in *Front*.

5 New CSG rendering algorithm

In this section, we present our new algorithm for CSG rendering. To justify its benefit over prior art, we revisit the speckle problem, which, as admitted by Hable and Rossignac [2], typically produces incorrect colors at a few pixels in most images generated by Blister (Fig. 2). To render a CSG model V, Blister peels the union of the boundaries of all the primitives of V treating them as a single surface S and trimming them against the Blist of V. This approach may lead to **speckles**. For example, consider the expression V=(A–B)+C shown in Fig. 13. A point X of A inside B is not contributing to the boundary of V. Yet, if it lies on the boundary of V, it will be classified by Blister as being ON/IN V and rendered. Although it has the correct depth (because there is a coincident point of C on the boundary of V), it may have a different normal (here the normal of A instead of the normal of C), which produces a shading mistake (speckle).

To solve this problem, we build on [33] and introduce a CSG rendering algorithm that peels and trims one primitive A at a time. Instead of trimming its boundary against the whole CSG expression, it trims it against the **active zone**, Z, of A. Z=WI–U is the intersection of the universe W with the **i-zone** I of A and with the complement of the **u-zone** U of A. I is the intersection of i-nodes and U the union of u-nodes defined as follows [18]. Consider the **path** from the root to A. The **i-nodes** of A are the children of intersection nodes of the path that are not in the path. The **u-nodes** of A are the children of union nodes of the path that are not in the path. Hence, Z is the intersection of W with each i-node and with the complements of each u-node of A. Note that, the CSG expression of the active zone of each primitive may be derived trivially from the CSG tree by a recursive traversal.

For example, in (A+B)(C(!D+E)) shown in Fig. 7, primitive A has one u-node, B, and one i-node, C(!D+E). Its active zone is !BC(!D+E). Thus, we can render the contribution of A as CST(A, !BC(!D+E)). E has two i-nodes, A+B and C, and one u-node, !D. Its active zone in is (A+B)CD. The contribution of E is CST(E,(A+B)CD).

Changes to A out of Z will not affect V. For instance, in our example, changing E in !D will have no effect on D–E. Changes of E in D will affect D–E, but will affect C(D–E) only if they are in C. Most importantly, the boundary of V is the union of the boundaries of each primitive P trimmed by its active zone Z. Based on this observation, our new CSG rendering peels each primitive while trimming it against its active zone Z and merges the results into a global z-buffer to select the front-most points.

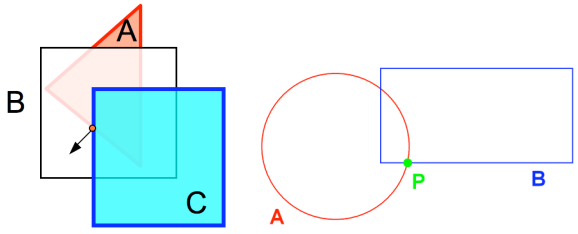


Fig. 13. A speckle on the boundary of A on (A–B)+C and a double point P on the boundary of A+B.

In this example, the final solid is rendered as the union of the following CSTs:

CST(A, C(!D+E)!B)
CST(B, C(!D+E)!A)
CST(C, (A+B)(D!E))
CST(D, (A+B)CE)
CST(E, (A+B)CD)

Rendering these primitive boundaries trimmed by their active zones will produce correct pictures when rendering opaque CSG models. In some applications, such as transparency or volume computation, it may be desired to ensure that overlapping portions of trimmed primitive faces (**overlap**) are not counted as multiple ON surfels.

We distinguish between speckles and overlaps. A **speckle** occurs when one point, which is outside its active zone, interferes with another point that should be rendered. An **overlap** occurs when two or more points that are ON the trimmed portions of different primitives coincide. Assume for instance that we have an overlap of two points. If neither is rendered, a hole will appear in the solid's boundary. If both are rendered, the overlap will appear twice more opaque, producing incorrect transparency images. To solve this problem, we choose an arbitrary ordering of the source primitives in a CSG expression. For two primitives L and R where L is ordered before R, L is considered slightly in front of R when two points from L and R overlap (share the same depth relative to the camera). For the right object in Fig. 11, we choose our ordering to be A, B, meaning that at P the point from A would be rendered and the point from B would not.

We can enforce this ordering during the ***Trim***() step of the CST algorithm. If a point of L is testing its parity against a solid R, L's parity is toggled if a point of R is behind or equal to L. In the reverse case, when R is evaluated against L, a point from R is toggled if a point of L is strictly behind the point from R. Effectively, the point from R is inside the solid L, whereas the point from L is outside the solid R.

6 Depth Peeling, Transparency, and Painting

Transparency is used to visualize the internal structure of a CSG solid or to show semi-transparent CSG parts as context when rendering their interferences. Correct rendering of transparency requires that we render the faces of the CSG model in front-to-back order and that we only render one candidate point per face of V for each pixel covered by it, even if several primitives overlap.

To ensure front-to-back order for transparency rendering, we peel the CSG model as follows. We render V as explained above and store the result in the front z-buffer F. Then, we render V again, but this time, we initialize the peeling of each primitive to the depth values stored in F. Hence, only candidate points behind F will be considered. Fig. 14 illustrates this process.

To ensure that we do not miss gaps or thin plates where surfels of two adjacent layers are close enough to share the same quantized depth, we treat the even and odd layers separately. To construct the first layer, we only render faces that would be front-facing in the resulting solid. Since our representation of the solid's boundary satisfies the parity

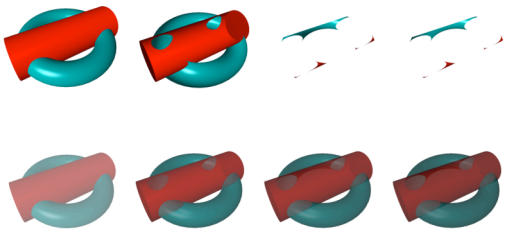


Fig. 14. The successive layers of a CSG solid are shown in depth order (top). They are merged for transparency (bottom).

test, the ray/boundary intersection points along the ray alternate between back-faces and front-faces. We can thus render the third layer as the second layer in the set of front-facing surfaces. More generally, layer 2k–1 is rendered as the k^{th} layer of the front-facing surfaces, and the layer 2k is the k^{th} layer of the back-facing surfaces. This transparent rendering method requires a buffer for front-faces and a separate buffer for back-faces. Since z-fighting between adjacent front and back faces is avoided, cracks, thin plates, and thin portions of the solid near sharp silhouette edges are rendered correctly.

We use transparency to visualize **interferences** as shown in Fig. 1.c. This is important in the inspection of mechanical assemblies and digital mock-ups for verifying that two distinct parts A and B do not overlap. While the interference can be visualized with existing algorithms by directly rendering AB, it helps to see the interference in the context of the A and B. In the case where A is yellow and B is blue, we can view the interference by creating two new primitives A′ and B′ which share the same geometry as A and B respectively, but are rendered with a red color. We then render the expression (A–B′)+(B–A′).

We can also use a CSG solid V to **paint** on a surface S, as shown in Fig 1.a. To do so, we render CST(S,V) in one color and CST(S,!V) in another.

7 Implementation and Optimization

We propose below three optimizations to improve the performance of CST rendering.

(1) When rendering opaque CSG models, we only peel&trim the **front-faces** of positive primitives and the back-faces of negative primitives.

(2) When rendering opaque CSG models, we need not trim the primitives against the **u-nodes** at all. Hence, we need only to use the i-nodes for trimming. If the extraneous portions that would have been trimmed away by u-nodes do not lie on V, they are guaranteed to lie in V, and hence, will be occluded by other CST faces of V. This optimization tends to considerably reduce the trimming time, especially in CSG models with many union operators at the top of the tree, which is often the case in CSG models of assemblies used in the automotive, naval, and aerospace industry.

(3) We can also **group leaf** nodes that share the same operator into similar nodes. The trimming step is based on the fact that we can evaluate a layer of pixels against a single primitive. We can also classify a layer against a subtree of primitives that all have the same parent operator, which

will be discussed in the next section. This optimization tends to reduce the number of stencil operations.

The Trim(V) operation in Section 5 refers to switching all stencil values in a buffer with the value of *src* to a value *dst*. We can perform this operation using the technique described by Hable and Rossignac [2], which inverts all bits that are different between *src* and *dst*. We perform this update by setting the following stencil states and rendering a rectangle over the entire screen. Thus, updating the stencil values after parity testing requires calling ChangeStencilState() twice, which requires two passes over the entire screen per primitive.

```
Trim( V )
    For each Primitive P in V
     Render V
     ChangeStencilState( (V.n << 1) | 0x1, V.t )
     ChangeStencilState( (V.n << 1) | 0x0, V.f )

ChangeStencilState( src, dst )
   glStencilMask( src ~ dst );
   glStencilFunc( GL_EQUAL, src, 0xFF );
   glStencilOp( GL_KEEP, GL_KEEP, GL_INVERT );
   DrawFullScreenRectangle();
```

We determine whether a candidate point X at pixel p is in an *intersection-group* G by scanning all of the primitives of G while using a *counter* [10] stored in the stencil of p. We initialize the *counter* to c_0. Then, for each positive primitive P of G, we rasterize the back faces of P while incrementing *counter* and the front faces of P while decrementing *counter* for each point of P in front of X. Note that processing P in this manner increments *counter* if $X \in P$ and leaves counter unchanged if $X \notin P$. Then, we do the same for each negative primitive N of G, but flip the role of front and back faces. Note that processing N in this manner decrements *counter* if $X \notin N$ and leaves counter unchanged if $X \in N$. Suppose that we had c_p positive and c_n negative primitives in G. $X \in G$ when X is in all positive primitives (i.e., *counter* was incremented c_p times) and in none of the negative primitives (i.e., *counter* was never decremented). Thus $X \in G$ when *counter*$=c_0+c_p$ Using DeMorgan's laws, we convert a union-group to an intersection-group by reversing the role of positive and negative primitives and switching the result.

If the depth complexity of each primitive is bounded by k (i.e., its boundary may intersect a line at 2k isolated points), *counter* may vary between c_0-c_n-k and c_0+c_p+k during this process. Hence, we select $c_0=c_n+k$ to ensure that *counter* stays positive and must allocate $\lceil \log_2(c_n+c_p+2k) \rceil$ stencil bits for it. Although this approach does not change the number of times each primitive P is rasterized during trimming, it reduces the number of stencil passes, which are the dominant fraction of the total cost because they visit all pixels of the screen. Instead of 2 stencil passes per primitive, the grouping approach requires 4 stencil-passes per group to reinitialize *count* after each group and then to perform the Blist logic, as explained earlier. Since this approach uses 4 stencil passes per group instead of 2 stencil passes per primitive, we achieve performance gains when the number of groups is less than half the number of primitives.

8 Results

To illustrate typical performance, we report for each model in Fig. 1 the number ***Prims*** of primitives and the rendering times (averaged over various viewing directions) for 4 methods: Blister (column ***Blister***); CST trimming against all branching nodes (column ***Z***); CST against i-nodes only (column ***I***); and CST against i-nodes using grouping (column ***G***). All times are reported in milliseconds on a Windows XP on a 3.2 Ghz Pentium 4 processor with an nVidia GeForce 6800 GT graphics card. The implementation uses OpenGL and Cg. All tests were performed in an 800×800 pixel window, with a view set so that the CSG solid barely fits in the window.

	Prims	***Blister***	***Z***	***I***	***G***
Complex	20	121	96	92	100
Gear	48	481	382	312	179

We observe the following. Performance of Blister for a given model varies considerably with the view orientation. For example, *Complex* takes twice as long to render from the side than from the front. Our CST-based algorithm is faster and less viewpoint-dependent. Performance of CST depends on the sum of the depth layers of its components, which varies less with view changes. In fact, it is constant if all primitives are convex. Its dominant cost lies in the stencil logic. Skipping u-nodes usually improves performance significantly. Grouping, which has an overhead, benefits only models, such as *Gear*, which yield large groups (intersection or union trees at the bottom of the CSG tree).

We also report the performance of the more advanced rendering techniques made possible by our CST approach. Since there is no other realtime algorithm that can produce correct images of semi transparent CSG models, we cannot provide a comparative analysis with prior art. *Bunny* refers to the Bunny with CST painted on it in Fig. 1. *Interference* refers to the rendering of the transparent solid with the interference in red in Fig. 1. *Transparent* refers to rendering the first 4 layers of the Complex solid in Fig. 1. *Inside* refers to showing the interior portion of the Complex object in Fig. 1. Prims is the total number of primitives in the scene, and CSTs is the total number of CST passes. For example, rendering a single layer of *Transparent* requires 20 CSTs, so rendering 4 layers requires rendering 80 CSTs. The reported times are in milliseconds.

	Prims	***CSTs***	***Time***
Bunny	2	2	110
Interference	2	4	80
Transparent	20	80	391
Inside	20	16	222

The rendering time for semi-transparent models is proportional to the depth complexity of the model, since each layer requires rendering each visible CST. The dominant rendering cost is the number of CSTs times the number of primitives that need to be evaluated for each CST.

9 Conclusion

We have presented a new, GPU-based, CSG rendering algorithm that fixes the shortcomings and extends the capa-

bilities of Blister. Our CST algorithm trims the boundary of each primitive against a Blist of its i-zone. We have shown that it outperforms Blister and eliminates speckles, which appear as several wrongly colored pixels in most Blister images. Furthermore, our CST approach provides the first solution for the realtime rendering of non-trivial semi-transparent CSG models. It also may be used for painting on solids, for highlighting primitive contributions in CSG models, and for representing and rendering trimmed faces of BReps without having to precompute their trimming curves, which are typically generated through slow surface/surface intersection calculations.

REFERENCES

[1] A. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys,* vol. 12, no. 4, pp. 437–464, 1980.

[2] J. Hable and J. Rossignac, "Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 1024–1031, 2005.

[3] A. Rappoport and S. Spitz, "Interactive Boolean Operations for Conceptual Design of 3-D Solids," *Proc. ACM SIGGRAPH,* pp. 269–278, 1997.

[4] J. Rossignac and A. Requicha, "Depth-buffering display techniques for constructive solid geometry," *IEEE Computer Graphics and Applications,* vol. 6, no. 9, pp. 26–39, 1986.

[5] J. Goldfeather, J.P.M. Hultquist, and H. Fuchs, "Fast constructive solid geometry display in the pixel-powers graphics system," *Proc. Annual Conference on Computer Graphics and Interactive Techniques*, pp. 107–116, 1986.

[6] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, "Near realtime CSG rendering using tree normalization and geometric pruning," *IEEE Computer Graphics and Applications*, vol. 9, no. 3, pp. 20–28, 1989.

[7] T.F. Wiegand, "Interactive rendering of CSG models," *Computer Graphics Forum*, vol. 15, no. 4, pp. 249–261, 1996.

[8] N. Stewart, G. Leach, and S. John, "Linear-time CSG rendering of intersected convex objects." *Proc. International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pp. 437–444, 2002.

[9] N. Stewart, G. Leach, S. John. "Improved CSG rendering using overlap graph subtraction sequences," *Proc. International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia,* pp. 47–53, 2003.

[10] S. Guha, S. Krishnan, K. Munagala, and S. Venkatasubramanian, "Application of the two-sided depth test to CSG Rendering," *Proc. Symposium on Interactive 3D Graphics, pp. 177-180, 2003.*

[11] J. Rossignac, "Processing Disjunctive forms directly from CSG graphs," *Proc. CSG 94: Set-theoretic Solid Modeling Techniques and Applications,* pp. 55-70, 1994.

[12] R. Banerjee and J. Rossignac, "Topologically exact evaluation of polyhedra defined in CSG with loose primitives," *Computer Graphics Forum,* vol. 15, no. 4, pp. 205–217, 1996.

[13] S. Kumar and D. Manocha, "Efficient rendering of trimmed NURBS surfaces," *Computer-Aided Design*, vol. 27, no. 7, pp. 509–521, 1995.

[14] M. Guthe, A. Balázs, and R. Klein, "GPU-based trimming and tessellation of NURBS and T-Spline surfaces," *ACM Transactions on Graphics,* vol. 24, no. 3, pp. 1016–1023, 2005.

[15] A. Rockwood, K. Heaton, and T. Davis, "Real-time rendering of trimmed surfaces," *Proc. ACM SIGGRAPH,* pp. 107–117, 1989.

[16] S. Kumar, "Preventing Cracks in Surface Triangulations," *Proc. Chimera 98: 4th Symposium on Overset Composite Grid & Solution Technology,* pp. 40–47, 1998.

[17] G.K. Cheung, R.W. Lau, and F.W. Li, "Incremental rendering of deformable trimmed NURBS surfaces," *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pp. 48–55, 2003.

[18] J. Rossignac and H. Voelcker, "Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms," *ACM Transactions on Graphics*, vol. 8, no. 1, pp. 51–87, 1988.

[19] J. Rossignac, "BLIST: A Boolean list formulation of CSG trees," Technical Report GIT-GVU-99-04, GVU Center, Georgia Institute of Technology, http://www.cc.gatech.edu/gvu/reports/1999/, 1999.

[20] H. Fuchs and J. Poulton, "Pixel-planes: a VLSI-oriented design for 3-D raster graphics," *Proc. Canadian Man-Computer Communications Conference*, pp. 343–347, 1981.

[21] D. Epstein, F. Jansen, and J. Rossignac, "Z-buffer rendering from CSG: The Trickle algorithm," Research Report RC15182, IBM Research, 1989.

[22] J. Rossignac and J. Wu, "Correct shading of regularized CSG solids using a depth-interval buffer," *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems, Eurographics Seminars,* pp. 117–138, 1992.

[23] C. Everitt, "Interactive order-independent transparency," Technical Report, Nvidia Corporation, http://developer.nvidia.com, 2002.

[24] A. Mamman, "Transparency and antialiasing algorithms implemented with the virtual pixel maps technique," *IEEE Computer Graphics and Applications,* vol. 9, no. 4, pp. 43–55, 1989.

[25] M. Kelley, K. Gould, B. Pease, S. Winner, and A. Yen, "Hardware accelerated rendering of CSG and transparency," *Proc. Conference on Computer Graphics and Interactive Techniques*, pp. 177–184, 1994.

[26] H. Du and H. Qin, "Integrating Physics-Based Modeling with PDE Solids for Geometric Design," *Proc. Pacific Conference on Computer Graphics and Applications,* pp. 198, 2001.

[27] B. Schmitt, G. Pasko, A. Pasko, and T. Kunii, "Rendering trimmed implicit surfaces and curves", *Proc. AFRIGRAPH*, pp. 7–14, 2004.

[28] B. Adams and P. Dutré, "Interactive Boolean operations on surfel-bounded solids," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 651–656, 2003.

[29] D. Liao and S. Fang, "Fast volumetric CSG modeling using standard graphics system," *Proc. ACM Symposium on Solid Modeling and Applications,* pp. 204–211, 2002.

[30] E. Jansen, "A Pixel-Parallel Hidden Surface Algorithm for Constructive Solid Geometry," *Proc. Eurographics*, pp. 29–40, 1986.

[31] G. Erhart, and R. Tobler, "General purpose z-buffer CSG rendering with consumer level hardware," Technical Report. VRVis 003, VRVis Zentrum für Vurtual Reality und Visualisierung Forschungs-GmbH, 2000.

[32] J. Ellis, G. Kedem, G.T. Lyerly, D. Thielman, R. Marisa, and J. Menon, "The Ray Casting Engine and ray representations," *Proc. ACM Symposium on Solid Modeling Foundations and Applications,* pp. 255–268, 1991.

[33] J. Rossignac, "CSG formulations for identifying and for trimming faces of CSG models," *Proc. CSG'96: Set-theoretic solid modeling techniques and applications,* pp. 1–14, 1996.

[34] R. Bryant. "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computing*, vol. 35, no. 8, pp. 677–691, 1986.

[35] R. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," *Proc. IEEE/ACM international Conference on Computer-Aided Design*, pp. 236–243, 1995.

[36] S. Akers, "Binary decision diagrams," *IEEE Trans. Computing*, vol C-27, no. 6, pp. 509–516, June 1978.

[37] B. Yang and D. O'Hallaron, "Parallel breadth-first BDD construction," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 145-156, 1997.

[38] H. Payne and W. Meisel, "An algorithm for constructing optimal binary decision trees," *IEEE Trans. Computing,* vol. 26, no. 9, pp. 905–916, 1977.

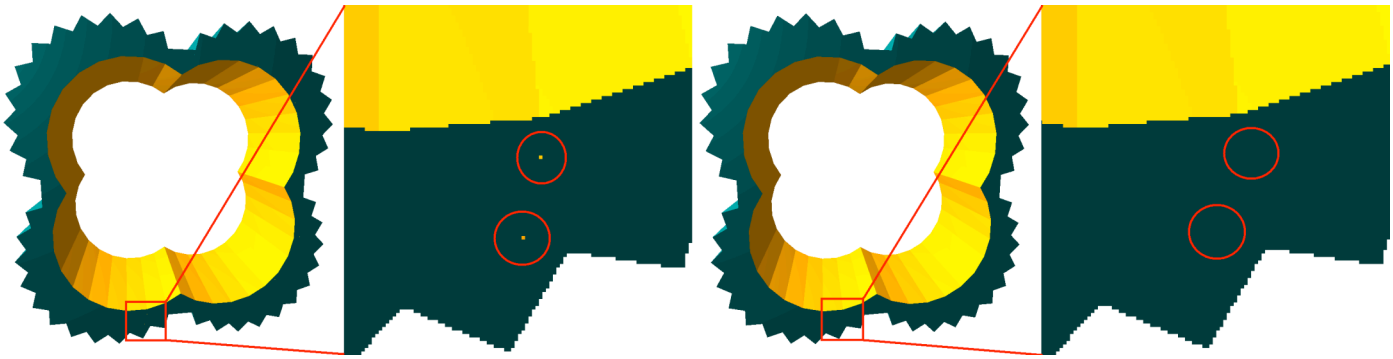
[39] J. Rossignac 2006, "Blist: Small footprint evaluation of Boolean expressions," Technical Report, GVU Center, Georgia Institute of Technology.

John Hable is a Software Engineer with the WorldWide Visualization Group at Electronic Arts. His research focuses on real-time photorealistic recreation of human performances and visualization of scenes defined by Boolean operators.

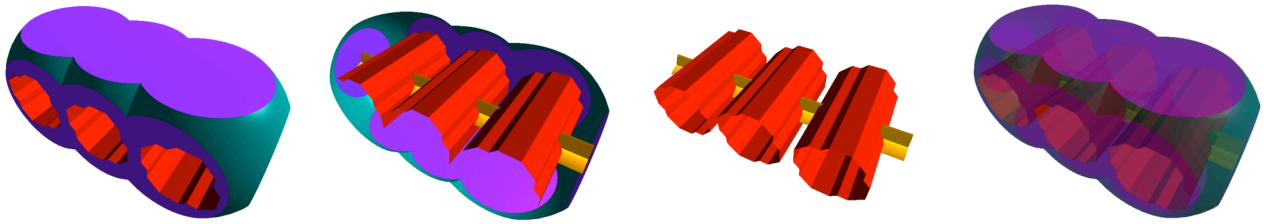
Jarek Rossignac is Professor of Computing at Georgia Tech. His research focuses on the design, analysis, compression and visualization of shapes and animations. Before joining Georgia Tech as Director of the GVU Center, he was Senior Manager and Visualization Strategist at the IBM T.J. Watson Research Center. He authored 19 patents and 120 articles, for which he received 5 Corporate and 8 Best Paper Awards. He created the ACM Solid and Physical Modeling Symposia series; chaired 20 conferences and program committees; and served on the Editorial Boards of 7 professional journals and on 52 Technical Program committees. He is a Fellow of the Eurographics Association.

APPENDIX: HIGH RESOLUTION IMAGES

Appendix Fig. 1: CST can be used for painting or carving on surfaces.



Appendix Fig. 2: Blister produces speckles (see magnification insert on the left). CST does not (right).



Appendix Fig. 3: CST images of the *Complex* CSG model (from left to right): Opaque rendering of the first depth layer (a); The second depth layer (b); Active parts of selected primitives (c); Semi-transparent boundary of the CSG model (d).