Direct Rendering of Boolean Combinations of Self-Trimmed Surfaces

Jarek Rossignac[a,1], Ioannis Fudos[b,*], Andreas Vasilakis[b,2]

[a]School of Interactive Computing, College of Computing, Georgia Tech, Atlanta, Georgia 30332, USA
[b]Department of Computer Science, University of Ioannina, GR45110 Ioannina, Greece

Abstract

We explore different semantics for the solid defined by a self-crossing surface (immersed sub-manifold). Specifically, we introduce rules for the interior/exterior classification of the connected components of the complement of a self-crossing surface produced through a continuous deformation process of an initial embedded manifold. We propose efficient GPU algorithms for rendering the boundary of the regularized union of the interior components, which is a subset of the initial surface and is called the trimmed boundary or simply the *trim*. This classification and rendering process is accomplished in realtime through a rasterization process without computing any self-intersection curve, and hence is suited to support animations of self-crossing surfaces. The solid bounded by the trim can be combined with other solids and with half-spaces using Boolean operations and hence may be capped (trimmed by a half-space) or used as a primitive in direct CSG rendering. Being able to render the trim in realtime makes it possible to adapt the tessellation of the trim in realtime by using view-dependent levels-of-details or adaptive subdivision.

Keywords: Rendering, CAD, trimming, GPU, capping, clipping, animation, solid modeling, CSG

1. Introduction

Rendering boolean combinations of self-crossing surfaces in realtime is central to a number of applications such as animation of deforming objects, preview of CAD operations and collision detection. FFD (free-form boundary deformations) is a popular paradigm for designing 3D shapes. FFD affords an intuitive direct manipulation and seems most appropriate for editing medical and artistic models or animations. The designer may for example use 3D input devices to grab, pull, and twist the 3D model in natural and predictable ways to create self-intersecting surfaces [13]. Unfortunately, FFD lacks a useful semantics of what happens when the designer wishes to create a self-intersecting surface model. One may argue that in a static model, the user could be asked to select which manifold portions of the surface should be removed by clicking on them. We offer a semantics that makes this selection process unnecessary. More importantly, if we want to apply these topological changes to an animated model, we cannot expect the user to perform these selections at each frame. We need a semantics for performing these selections automatically in a manner that is coherent over time and that is compliant with the results that would be obtained through CSG operations. This paper introduces a framework for treating self-trimmed surfaces as first class citizens, allowing to use them as CSG primitives or to show their cross-sections (intersections with a plane) using capping. In this work, we formally define the problem, explore rules that capture application semantics and provide efficient GPU-rendering algorithms.

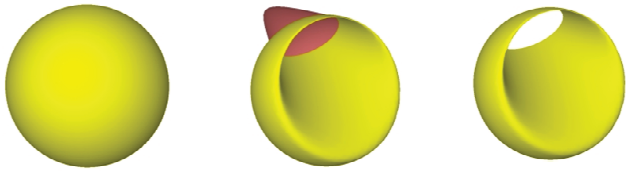


Figure 1: We show the original manifold boundary S_0 (left), a frame S_t produced by continuously deforming S_0 (we use pink to illustrate the part of the surface that should be trimmed) (center) and the trimmed result $T(S_t)$ (right).

Formally, a surface S is **manifold** when it is a compact and orientable two-manifold without boundary. We say that S is **self-crossing** when its immersion contains non-manifold self-intersection edges where S "passes through itself", as shown in Figure 1. Hence, two or more different points of S coincide at each point of a self-crossing curve of the immersion. We say that S is a **boundary** when S is the boundary of some solid (closed-regularized point set) that we denote *I(S)* and call the **interior solid** of S.

Consider an initial manifold boundary S_0 that is not self-crossing and a continuous process D_t that deforms this surface while keeping it an immersed sub-manifold. Let S_t denote the instance $D_t(S_0)$ of the deformed surface at time t. Assume that S_t is self-crossing, then we say that S_t is a **Self-Crossing Surface**, abbreviated *SCS*, i.e., a compact, immersed, and ori-

*Corresponding author
Email address: fudos@cs.uoi.gr (Ioannis Fudos)

[1]The material is based on work supported by the National Science Foundation under Grant No 0811485.

[2]Research of this author has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

entable surface with transverse self-intersections. An example of the above setting is illustrated in Figure 1.

We explore here different semantics for defining the solid that it represents, which we call its interior $I(S_t)$, and hence also its **trim**, $T(S_t)$, which is the subset of S_t that is the boundary of $I(S_t)$. We say that $T(S_t)$ is a **Self-Trimmed Surface**, abbreviated *STS*. The interior along with the STS define a manifold or a non-manifold object.

In this context we identify two generic problems and devise rules to capture them:
Problem I: Given an SCS (Figure 1(b)) determine the trim (Figure 1(c)).
Problem II: Given an initial manifold boundary S_0 (Fig. 1(a)) and a continuous process D_t that deforms this surface to an SCS S_t (Figure 1(b) determine the trim $T(S_t)$ (Fig. 1(c)).

To address the first problem, we explore static rules that depend only on the SCS and evaluate at least in simple cases how well the results they produce match what we consider to be plausible intentions of the designer.

The second problem corresponds to dynamic rules that depend on the deformation history and the SCS. We are particularly interested in formulations of $T(S_t)$ that correspond to a designer's intuitive expectation of the sequence of results that should be produced by a reasonable deformation D_t that creates several self-crossings. In particular, we propose semantics that mimic locally the natural behavior of incremental Boolean operations, where self-crossings are created in S_t one at a time and each performs a local union or intersection of shapes defined partially by two portions of the previous frame S_{t-1}.

We also introduce practical and efficient GPU-based **trimming** algorithms that render $T(S_t)$ directly by scan-converting S_t and S_{t-1} (where t is the frame number) without the need for computing self-intersection curves. We do this by testing surfels, to establish whether they lie on the boundary of $I(S_t)$. Surfels are represented by fragments of the surface that arise from the intersection of the surface with a ray originated at the center of the corresponding pixel of the viewing plane.

We claim three advantages of such a direct trimming and rendering approach. The first advantage is the elimination of the cost of computing self-intersection curves and of identifying the faces (connected components that are cut out by these curves). Such a cost would otherwise make it impossible to render the trim during deformation animations or perform interactive editing. The second advantage is the flexibility of being able to define S_t as the result of a (possibly adaptive) subdivision process to be carried out on the GPU. Finally, we can render on the GPU the result of combining the interiors of two or more self-crossing surfaces through CSG operations.

2. Background and Problem Definition

An CS S partitions the 3D space W into open full dimensional **components** C_i (i.e., the maximally connected components of $W - S$), one of which is infinite (denoted here by C_0). Each component is classified as either ***interior*** (also denoted as ***in)***, i.e., part of the interior of the solid, or ***exterior*** (also denoted as ***out)***. The solid $I(S)$ represented by S is the closure of the union of all interior components. Our objective is to define a rule that selects the components of $I(S)$. To obtain a bounded solid, C_0 should not be included, and hence is **classified** as *out*. Other components may be classified as *in* or *out*, depending on the chosen rule.

The **trim**, $T(S)$, is the boundary of $I(S)$. Hence, trimming amounts to discarding portions of S that separate either two interior or two exterior components.

Various **rules** (semantics) may be used to associate a solid $I(S)$ with an SCS S. One may conceive interesting rules that compute new bounding surfaces for solid $I(S)$ (by for instance using the convex hull of S or a visibility graph). Here, we focus on rules that have the **boundary diminishing** property, which states that the boundary $T(S)$ of $I(S)$ must be a subset of S. Note that this property is satisfied by Boolean and regularized Boolean operations [26, 20].

In 2D, the **index** (also called **winding number**) $w(p, C)$ of an oriented, closed-loop, self-crossing curve C around a given point $\boldsymbol{p}$ that is not on C is an integer representing the total number of times the curve travels counter-clockwise around the point. The winding number depends on the orientation of the curve, and is by convention negative if the curve travels around the point clockwise. All points in a given component (maximally connected component of the complement of C) have the same winding number. The infinite component has winding number 0. One may easily keep track of the winding number by propagating it from one component to an adjacent one. Crossing the curve once increments or decrements the winding number, depending on the orientation of the curve relative to the direction of the crossing.

In three dimensions, the **index** $w(p, S)$ of a point p with respect to an SCS S may be defined as follows. We assume that p is not on S. Consider any **path** P from infinity to p. Let k_i be the number of times that P enters S (i.e., crosses the boundary in a direction opposite to the outward normal) and k_o be the number of times that P **exits** S (i.e., crosses the boundary in a direction confluent to the normal). Then, $w(p, S) = k_i - k_o$. Figure 2 shows an SCS cross section (green self-crossing polyline) with the triplet $(k_i, k_o, w(p, S))$ indicated for each area, where k_i and k_o were computed from the left. Note that points of the same component may have different k_i and k_o (for example when counting from the bottom) but they have the same index.

For conciseness, we denote $w(p, S)$ by $w(p)$ or by simply w. Heisserman [8] provides an equivalent definition of the index (winding number) as the number of times the surface encloses a point.

Note that in situations where P simultaneously crosses several neighborhoods of S, the crossing of each neighborhood must be accounted for separately.

The index is the signed generalization of the *overlap count* which is defined as the unsigned count of the number of surfels that correspond to a pixel and is used in rendering to determine the transparency effect. Note that the index is in general not equal to the overlap count, which is defined at a point p as the number of times a specific ray from p to infinity hits the surface. When the ray is aimed at the viewpoint, the overlap count may

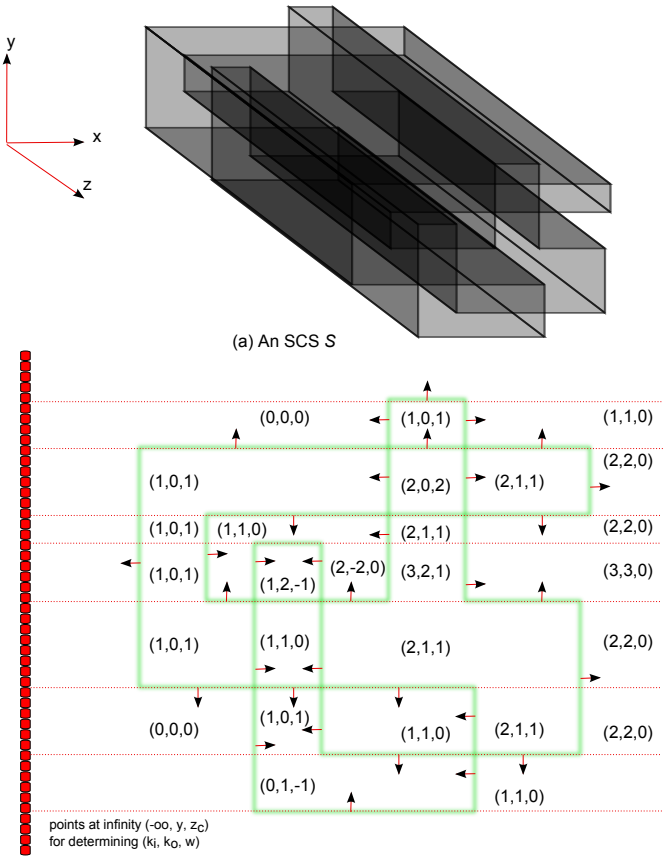


Figure 2: (a) An SCS S. (b) A cross section of S on plane $z = z_c$ illustrated with thick green transparent line with normal vector orientation marked. We indicate the values of the triplets (k_i, k_o, w) using $(-\infty, y, z_c)$ as point at infinity for determining the characterization for a point $p(x, y, z_c)$. The horizontal dashed red rays originating from the points at infinity along with the green polyline partition the plane into areas with the same triplet value. Note that k_i, k_o depend on the selection of the point at infinity, whereas the winding number is independent of this choice.

be computed on the GPU for each pixel and used to control transparency effects. Note that the parity of the overlap count is independent of the direction of the ray, identical to the parity of the index, and constant throughout a component.

We want to use the fragments (depth, color, and normal values of S associated with points of T that project on a pixel center) that are generated by this rasterization to render scenes where $I(S)$ is **capped** [14, 23] and combined with other shapes through CSG operations [6, 7]. **Capping** returns the intersection of $I(S)$ with a 3D region R that is the intersection or the union of (usually) linear half-spaces and displays the **caps**, i.e., the portions of the boundary of R in $I(S)$.

3. Prior Art

Interior Visualization. Different techniques have been proposed for visualizing the interior and hidden portions of solids or assemblies. Hidden edges and silhouettes may be overlaid with a shaded rendering of the visible surfaces [24]. Surface transparency and depth ordering [1] may be used to modulate the color based on the translucent surfaces seen through a pixel. Volume rendering may be used to modulate color based on the thickness of solid layers traversed by a ray from the viewpoint (see e.g. [29]). A portion of the boundary may be removed using clipping planes or solids, exposing hidden surfaces and potentially back-facing portions of the boundary. Because such images may appear confusing to the casual viewer, capping [23] is used to render the intersection of the boundary of the clipping region with the interior of the solid. Capping uses the parity of overlap count (or equivalently of the index) to decide whether a candidate point on the boundary of the clipping region lies inside the solid to be capped. This parity may be computed efficiently at each pixel using a stencil bit. The self-trimming solution proposed here builds upon the capping approach, but differs from it in two important ways: (i) candidate points are generated on the SCS itself (not on the boundary of a trimming region) and (ii) a different test is used to classify candidate surfels.

Direct CSG Rendering. A complex solid may often be designed as a Boolean combination of other solids. The design sequence may be captured as a Constructive Solid Geometry (CSG) representation [20] that defines a Boolean expression with union, intersection, and difference operators and with primitive solids as operands. In early solid modeling systems [28] the primitives were restricted to simple quadrics (block, sphere, cone, cylinder and torus). Computing the boundary of solids defined in CSG with more general primitives (triangle meshes, NURBS, subdivision surfaces) is computationally expensive and numerically delicate. To address this problem, several screen-based techniques have been proposed for rendering CSG models directly on the GPU by classifying surfels against each primitive. Some approaches [3, 5, 9] use a disjunctive form (union of intersections) formulation of the Boolean expression to simplify the trimming. Unfortunately, the number of intersections (products of a disjunctive form) may grow exponentially with the number of primitives. To avoid this complexity exposure, the Constructive Solid Trimming (CST) [7] trims the boundary of each primitive against the Blist [6] of its active zone [25], which defines the solid where the boundary of the primitive contributes to the boundary of the solid. Recently, Rossignac has shown that 6 stencil bits per pixel suffice for rendering arbitrarily complex CSG models, by providing a linear cost algorithm that swaps left and right operands of the Boolean expression of n literals so that it may be evaluated using $O(loglogn)$ space [21]. Recently, Zhao et al. [31] have introduced a fast hardware assisted method for approximately reconstructing CSG results using rasterized views. These methods are not capable of handling self-trimmed surfaces.

The self-trimming approach proposed here fits well with these direct CSG rendering approaches and permits the use of CSG primitives defined by self-trimmed surfaces. Because the trim can be evaluated at each pixel directly from the representation of the SCS, the SCS may evolve from one frame to the next, as the result of a change in the subdivision depth (dynamic level-of-detail) or as the result of a free-form deformation or

skeletal bending during an animation.

Self-trimming. Static interior classification of self intersecting curves has been studied extensively in 2D (see for example [8]). When the bounding loops of several connected regions of the plane evolve over time and self-overlap, one may wish to decide automatically which region should be visible at each pixel. The union of the boundaries of these regions decomposes the union of the regions into cells. A valid association of each cell with a unique region must satisfy constraints that ensure that the arrangements can be realized by placing and interleaving physical cut-out regions on a plane [16].

The problem becomes more complex when these regions are connected, and hence can no longer be differentiated. In particular, several authors have addressed the problem of defining the interior of self-crossing curves of a particular class, which may be characterized by stating that the curve is the boundary of a topological disk that does not contain any fold-over (i.e. each portion of that disk is front facing). The issue of deciding whether a self-crossing curve is in this class and whether it is the projection of the boundary of a front-facing disk in 3D has been discussed by [2]. A simple approach that works well in practice has also been proposed by [18].

Efficient Multi-fragment Rendering. Correct rendering and trimming of self-crossing manifolds requires per-fragment sorting operations, which can be more expensive than sorting object (or geometry) primitives, but can correctly handle intersecting or overlapping geometry. Maule et al. [15] classify the fragment-sorting methods into two categories, *buffer-based* methods which first store and then sort the fragments and *depth peeling* methods that extract depth order implicitly through multi-pass rendering.

Liu et al. [12] introduced CUDA implementations for dynamically and efficiently building a fixed data structure, called *FreePipe*, maintaining many fragments per pixel in real-time. The major limitations of FreePipe is the potentially large memory requirement depending on the screen resolution and depth complexity of the scene. Furthermore, such techniques usually require to switch from the traditional graphics pipeline to a CUDA rasterizer scheme.

To alleviate high space cost, Yang et al. [30] propose a dynamic construction of a per-pixel *Linked-List* structure aiming at avoiding unnecessary memory pre-allocation. However, it is slower than previous methods due to the excessive access of the shared memory. This technique requires the most recent APIs (*OpenGL* 4.2 or *DirectX* 11) supported only on recent graphics processors.

Multi-pass rendering has been used to carry out many effects, often substantially limiting performance. Probably the most well-known multi-pass technique is *front to back peeling* [4] which works by rendering the geometry multiple times, peeling off a single fragment per pass. *Dual depth peeling* [1] reduces the number of geometry passes by capturing both the closest and farthest fragments in each pass. For a comprehensive survey on the complete raster-based fragment-sorting techniques readers may refer to [15].

In this work, we have adapted several state of the art multi fragment techniques to achieve efficient rendering of self-trimmed surfaces (Front to back, Dual depth peeling, FreePipe, LinkedLists) and we have provided comparative results in terms of time and space requirements.

4. Revisiting Interior Exterior Classification Rules

Previously proposed rules for interior classification are static and sometimes depend on the topology of the self-intersecting surface (for example index based classification). Such rules do not capture the process of dynamically extending interior or exterior parts in a consistent and intuitive manner (*Problem II*) but may yield useful classifications for determining the interior of an SCS of unknown origin (*Problem I*).

4.1. Static Rules

Static rules are usually based on the point index with respect to the current SCS for classifying points as interior or exterior.
We wish for the classification rules to be **intrinsic**, i.e., independent of the choice of coordinate system. In some applications, it is useful to support rules based on global topological or integral properties (such as the genus or volume of a component or its surface area) [19]. We will discuss possible extensions of our work to support such **global rules**, but this section is primarily focused on local rules. A rule is **local** if the classification is based on the intersections of S with a given ray that does not intersect any curve derived by a self-crossing. In this section, we consider only static rules that are based on the index w of the component that we wish to classify.

The popular **parity** (also called *alternating interior*) rule classifies a component as *interior* when the index of its points is *odd*. This corresponds to switching the *interior* status each time one traverses S. Note that the result is not altered by a global change of the orientation of S. Hence, we say that the parity rule is **orientation invariant**. Unfortunately, the parity rule will only trim (remove from S) some of the two-dimensional self-overlapping portions of S. Thus, if S is self-crossing, but does not have self-overlaps, then $T(S) = S$. Although this is a useful rule, it does not allow the designer/user to easily modify the genus of the interior closure of a manifold boundary S by warping S so that it crosses itself. When S overlaps itself (along a full-dimensional portion instead of cleanly self-crossing), the parity of the number of portions that overlap at a point $\boldsymbol{p}$ of S defines whether $\boldsymbol{p}$ is in $T(S)$.

Unfortunately, the alternating interior rule is rarely an acceptable option since

1. Usually it does not trim the surface at all, hence it yields solids with $S = T(S)$ (except in self overlapping parts).
2. It produces non-manifolds solids where each self crossing edge is a non-manifold edge.

Both facts are illustrated in 2D in Figure 7(b). The red blurred lines indicate the trim (i.e. the STS which is the surface after trimming). In the simple case of Figure 9((left) all three manifold subparts are considered inside with the parity rule, even the clearly negative volume in the middle. Finally, they are connected through shared curves resulting in a non-manifold object.

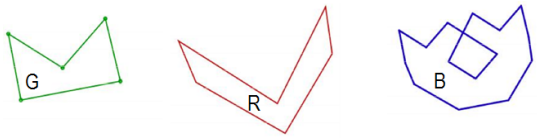


Figure 3: An example where the positive index rule works.

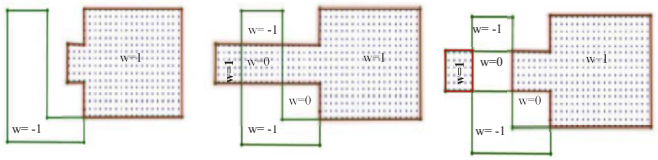


Figure 4: The green self-crossing loop (left) defines two regions, one of which has negative index and is discarded. Then we grow the other region (which is the interior defined by the self-crossing loop) by extruding a portion of its left border so that it overlaps the discarded region. We expect the space conquered by this extrusion be part of the new interior (center). The red line indicates the trim $T(S)$ (the boundary of the interior). Note that using the positive index rule (right) does not produce the expected result.

Heisserman [8] has proposed the **positive index rule** *(1-st unary intersection)* that classifies as *in* the components for which w is positive (*the closure of the set of points with winding number greater or equal to 1*). Of course, w is normally 1 inside S and 0 outside when S is free from self-intersections. This rule is not orientation invariant. This semantics is appropriate for applications that involve growing an initial set through offsetting [22], sweeps and Minkowski sums [10]. For example, Figure 3 shows a green loop with interior G, a red loop with interior R, and a self crossing blue loop defining an interior B, which is the Minkowski average $\frac{G \oplus R}{2}$ [10], where the interior is defined as the set of points with strictly positive winding number.

Defining the interior by $w \geq 1$ works well for cases such as the ones shown in Figures 7(a) and 8 but does not yield intuitive results for the case of Figure 4. Also it may improperly classify a region, such as component ***e*** in Figure 9. Finally, to derive the complement of the solid under the positive index rule, we should both change the orientation of the surface and adjust the index by adding a surrounding box (i.e. assign index $-w + 1$).

Here we propose the **alternating border rule** where a point p is *in* if and only if $\left\lceil \frac{w(p,S)}{2} \right\rceil \%2 = 1$. In simple configurations, it is equivalent to the positive index rule as shown in Figure 8. It has however two interesting and intuitive properties.

When the surface is free from self-overlaps, but crosses itself, then the classification of S, as being part of the trim T of $I(S)$ or not, alternates at each crossing edge. This is illustrated in 2D in Figure 5, which shows that this simple rule makes it easy to design and represent faces that are simply connected by using a single loop. In other words, the union of the self-crossing curves decomposes S into faces. This is substantiated by the following Theorem.

Theorem 1. *Adjacent faces (those incident upon the same self-crossing edge) have opposite classification with respect to the trim T (alternating border).*

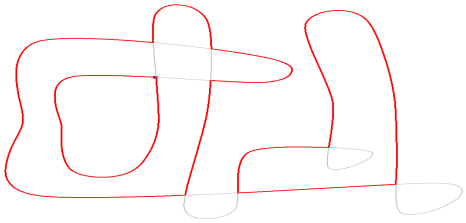
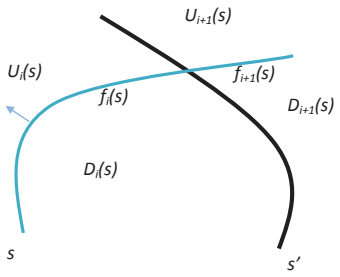


Figure 5: An example of creating a genus-1 object by deforming a single loop.

Proof. Consider a surface part s crossing a surface part s' and let $f_i(s)$ and $f_{i+1}(s)$ be the two adjacent faces. We shall prove that $f_i(s)$ is part of the trim if and only if $f_{i+1}(s)$ is not part of the trim (see Figure 6). For the purposes of the proof we will use an equivalent definition of the alternating border rule: a point p is *in* if and only if $w(p)\%4$ is 1 or 2. Equivalently, p is *in* if and only if there is an integer λ such that either $w(p) = 4\lambda + 1$ or $w(p) = 4\lambda + 2$. Clearly, p is *out* if and only if there is an integer λ such that $w(p) = 4\lambda$ or $w(p) = 4\lambda + 3$. Assume $f_i(s)$ is part of the trim, then the adjacent components $U_i(s)$ and $D_i(s)$ will have classification *in/out* or *out/in* respectively. Since the components are adjacent to the face their index numbers will differ by one. Let $U_{i+1}(s)$ and $D_{i+1}(s)$ be the adjacent components to face $f_{i+1}(s)$. Without loss of generality assume $w(D_k(s)) = w(U_k(s)) + 1$ for $k \in \{i, i+1\}$. Then for every case of $U_i(s)/D_i(s)$ being *in/out* or *out/in* $U_{i+1}(s)/D_{i+1}(s)$ have classification *out/out* or *in/in* (see Figure 6). Therefore, $f_{i+1}(s)$ is not part of the trim. Vice versa, suppose $f_i(s)$ is not part of the trim. Likewise, it follows (see Figure 6) that $f_{i+1}(S)$ is part of the trim. □



adjacent component *classification for face $f_i(s)$. Components $U_i(s)$ / $D_i(s)$*	index numbers w($U_i(s)$) / w($D_i(s)$) w($U_i(s)$)+1=w($D_i(s)$)	Index numbers w($U_{i+1}(s)$) / w($D_{i+1}(s)$) case w increases by 1 or case w decreases by 1 after crossing s'	adjacent component *classification* for face $f_{i+1}(s)$ (after crossing s'). Components $U_{i+1}(s)$ / $D_{i+1}(s)$
in/in	4λ+1 / 4λ+2	4λ+2/4λ+3 or 4λ/4λ+1	in/out or out/in
out/out	4(λ-1)+3 / 4λ	4λ/4λ+1 or 4(λ-1)+2/4(λ-1)+3	out/in or in/out
out/in	4λ / 4λ+1	4λ+1/4λ+2 or 4(λ-1)+3/4λ	in/in or out/out
in/out	4λ+2 / 4λ+3	4λ+3/4(λ+1) or 4λ+1/4λ+2	out/out or in/in

Figure 6: Establishing that under the alternating border rule, for adjacent faces it holds that exactly one of them will be part of the trim. A face is part of the trim if the two adjacent components have different interior/exterior characterization (*in/out* or *out/in*).

Furthermore, the alternating border rule does not generate

non-manifold edges unless the surface crosses itself multiple times along the same intersection curve.

Lemma 1. *The alternating border rule does not produce non-manifold solids with simple self-crossings (see Figure 7(c))*

Proof. Consider 4 portions of the surface incident upon any segment of a self-crossing intersection curve C. Two of these are trimmed away, because our rule toggles trimmed/retained classification when crossing C. Hence, the segment has two incident portions and is thus manifold. □

Finally, the alternating border rule has two more practical characteristics:

1. We may obtain the complement of a solid, simply by adding two bounding boxes with the same surface orientation (index becomes $w-2$ or $w+2$). This follows directly from the equivalent definition of the alternating border rule in the proof of Theorem 1.
2. If we reverse the surface orientation we obtain the complement of the trim. This follows immediately from Theorem 1.

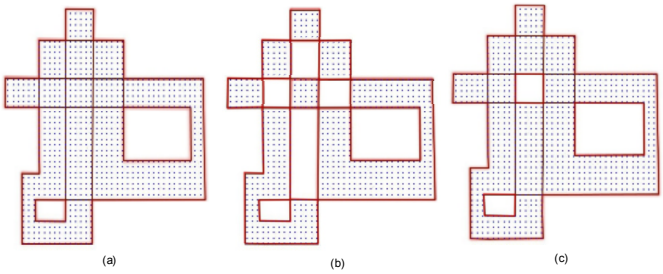


Figure 7: Interior classification using (a) the positive index rule, (b) the alternating interior rule and (c) the alternating border rule.

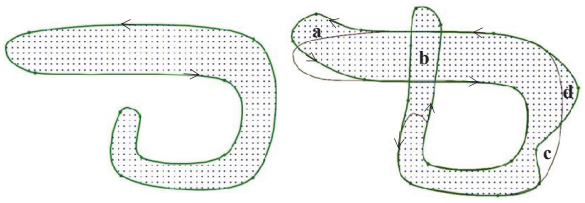


Figure 8: An example where the positive index number rule derives intuitive interior/exterior classification: (a) top tip wagging (b) bottom tip extending (c) dent creation (d) bump creation.

4.2. Dynamic Rules

Figure 9 illustrates the limitations of static rules with respect to *Problem II*. We use a deformation of an initial self-crossing curve shown on the left that extends the bottom tip upwards. The intuitively correct result is shown in the right. However, both the positive index rule and the alternating boundary rule would classify area ***e*** as exterior since its index is 0.

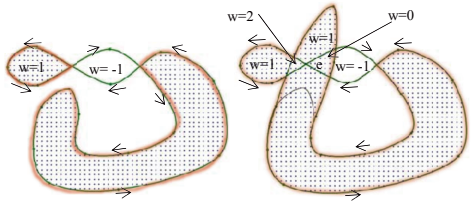


Figure 9: A deformation example where static rules fail to derive intuitive interior/exterior classification. The initial self-crossing curve (left) is modified (right) by extending the tip of the bottom part upwards. This change creates a region e where $w = 0$ and which is hence excluded by the static rules. Yet, intuitively, it should be part of the interior, since it corresponds to Boolean union of the initial interior with the extruded region.

To address this problem, we propose more complex rules, which at each stage, compare the previous and current indices of each region. More formally, we define a characterization of the components $C_i^{'}$ at some stage of the deformation process based on the current index of the component, and the classification of the component and the index with respect to the surface S at the previous stage of the deformation.

Formally, components $C_i^{'}$ are formed by a deformed surface $S^{'}$ that is derived by performing k **disjoint concurrent deformations** on S, such that $S^{'} = f(S) = f(s_1) \cup f(s_2) \cup \ldots \cup f(s_k)$, where $\{s_1, s_2, \ldots, s_k\}$ is a partition of S and for all surfaces in $\{f(s_i) : f(s_i) \neq s_i\}$ it holds that they do not cross, self-cross, overlap or self overlap. This restriction ensures that a boundary surface may cross a point only once during each set of concurrent deformations. The k concurrent disjoint deformations may be applied in a number of steps called frames. Determining the interior/exterior is based on the classification and point index obtained for the reference surface S and the new point index with respect to the current surface $S^{'}$.

4.2.1. Properties Characterizing the Behavior of Dynamic Rules

We will determine whether the following properties hold for post-deformation interior/exterior classification semantics:

Extension-normal confluence property. When deforming a surface by displacing it locally in the direction of the outward pointing normal, the points crossed either become interior or are not affected.

When we deform the surface in the opposite drection opposite to the normal the points crossed either become exterior or are not affected. Points whose index increases are candidates to become interior points and points whose index decreases are candidates to become exterior points. Points whose index does not change preserve the status that they had before the deformation.

Complement symmetry property. If we apply the same deformations on the complement we obtain the complement of the result.

Component homogeneity property. Each component contains only interior or only exterior points. This is a very important property since otherwise the border of the interior parts may not be a subset of the deformed initial surface. This is

equivalent to the aforementioned ***boundary diminishing*** property.

The following rules are based on the point index variation and the interior/exterior classification of the reference surface S. We denote the previous and the current index at a point p by $w(p, S)$ and $w(p, S')$, where S and S' denote the surface before and after the deformation, and the previous and the current interior/exterior classification by $i(p, S)$ and $i(p, S')$ respectively. Let $i(p, S)$ be the classification of point p with respect to surface S. Then, $i(p, S)$ is 1 when p is in, 0 when p is out, and undefined when p is on $T(S)$.
For the initial classification of the SCS surface S_0 prior to any deformation, we use the alternating border rule. However, another scheme, such as the positive index rule, could be used if desired.

4.2.2. Constructive Rule

Here we define the constructive rule that emulates CSG behavior among the original solid and the newly created volumes due to deformation.

According to CSG we employ additive (union), subtractive (difference) and intersection deformation semantics for interior/exterior classification. After each step of concurrent deformations, we determine the interior/exterior classification of a point p with respect to the deformed surface S' by performing a union, subtraction or intersection between the original solid S and the newly created volumes. A point belongs to the newly created volume if and only if its index has been modified, i.e. $(w(p, S)! = w(p, S'))$

$$i(p, S') = i(p, S)\, op\, (w(p, S)! = w(p, S'))$$

where op depends on the type of deformation. For additive deformation $AopB$ corresponds to logical OR ($A \vee$B), for intersection deformation it corresponds to logical AND ($A \wedge$B) and finally the subtractive operation $AopB$ is realized as $A \wedge \neg B$

Additive deformation corresponds to adding a part to the interior (set union), subtractive deformation corresponds to subtracting a part from the interior (set difference). These semantics yield results that are symmetric to the complement if we replace each additive with a subtractive deformation and vice versa.
We observe that although this rule captures design intent and has a constructive nature, it does not preserve component homogeneity. Thus, in some cases this rule may yield highly non intuitive results for users not familiar with the CSG process. For such users, results where the trim is not part of the initial surface S (see Figure 10) may look ill-defined. To address this problem, we use the confluent deformation rule (see Figure 10(f)).

4.2.3. Confluent Deformation Rule

After each step of concurrent deformations, the interior exterior classification is determined by the following formula:

$$i(p, S') = \begin{cases} i(p, S),\ w(p, S) = w(p, S') \\ \left\lceil \frac{w(p,S') - w(p,S)}{2} \right\rceil \%2,\ otherwise \end{cases}$$

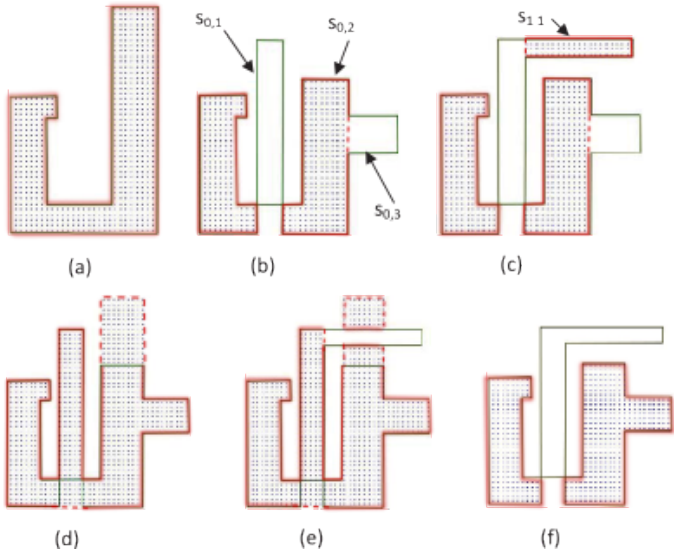


Figure 10: Illustrates how the constructive and the confluent deformation rules work. Interior parts are shown as shaded regions. The SCS is depicted by the green polyline. Red blurred lines indicate the trim (i.e the border of the new solid). Dashed red lines indicate parts of the trim that are not part of the SCS. (a) The original object and boundary, (b) after applying a set of 3 concurrent disjoint deformations f on surface parts $s_{0,1}$, $s_{0,2}$ and $s_{0,3}$ on (a) with subtractive semantics (constructive rule), (c) after applying a set of one deformation g of $s_{1,1}$ on (b) with additive semantics (constructive rule), (d) after applying f on (a) with additive semantics (constructive rule), (e) after applying g on (d) with subtractive semantics (constructive rule), and (f) after applying f on (a) and then g using the confluent deformation rule. In cases (a)-(e) the boundary of the shaded regions (trim) is not always a subset of the SCS.

which turns out to be equivalent to:

$$i(p, S') = \begin{cases} i(p, S),\ w(p, S) = w(p, S') \\ 0,\ w(p, S) > w(p, S') \\ 1,\ w(p, S) < w(p, S') \end{cases}$$

Note that, given that if $w(p, S')$ has changed then it differs from $w(p, S)$ only by one. Thus, $i(p, S')$ is 1 if and only if the point index is increased (extending the interior) and 0 if and only if the point index is decreased (extending the exterior). Figure 10(f) illustrates the result of applying two sets of concurrent disjoint transformations on the object of Figure 10(a).

4.2.4. Homogeneous Confluent Deformation Rule

The confluent deformation rule can be extended so as to enforce component homogeneity by imposing the following restriction:

A part s_{out} of surface S that is not part of the border B cannot be deformed towards the normal if s_{out} is between two exterior components. Likewise a part s_{in} of a surface S that is not part of border B cannot be deformed in a direction opposite to its normal if s_{in} is between two interior components. This restriction improves the confluent deformation rule semantics by enforcing component homogeneity.

To enforce the deformation restriction we need to detect trimmed off parts of the surface, i.e. parts that do not belong to the border and the interior/exterior classification of the adjacent

components. To simplify user interaction we suggest to prohibit deformations of the surface parts that have been trimmed off.

Table 1 provides a comparative overview of the interior/exterior classification rules presented in this section based on their principle, the efficiency of their implementation, their properties and their intuitiveness with respect to graphics and CAD designers. Graphics designers expect continuity during deformation sequences and consistency as far as viewing from outside is concerned. On the other hand, CAD designers expect robustness, manifold objects and are better acquainted with solid modeling operations.

Implementing these rules efficiently in hardware is far from trivial. We need to detect fragments that belong to the trim of the surface based on the current index, the reference frame index and the interior/exterior classification. In all cases we shall maintain in the current scene the fragments of the reference frame as well. By doing so, we can build efficient algorithms for realizing the confluent deformation rules with or without component homogeneity (see Section 5).

Table 1: Comparative overview of the properties and characteristics of static and dynamic rules

Rule	*based on*	*intuitiveness for CAD designers*	*intuitiveness for graphics designers*	*Properties*			*Implementation efficiency*
				component homogeneity	*extension-normal confluence*	*complement symmetry*	
Static rules	index	low	low	√	NA	NA	very efficient two pass
Constructive	index change and previous classification	high	low	-	-	√	efficient multipass or freepipe
Confluent deformation	index change and previous classification	moderate	moderate	-	√	√	efficient multipass or freepipe
Homogeneous confluent deformation	index change, previous classification	moderate	high	√	√	√	efficient multipass or freepipe

5. The Rendering Algorithm

In this section we present rendering algorithms for self-crossing manifolds and in particular we explain how to compute efficiently the point index, and how to perform efficiently trimming, clipping, capping and CSG operations. We discuss how the static and the dynamic rules are realized in this context. For computing and rendering the trim, we have employed multipass (sort-independent) and buffer-based peeling techniques. The basic process underneath all these techniques is the same: process all fragments per pixel (in addition to reference frame information for the dynamic rules) to determine the index and the interior/exterior classification. Section 6 presents results using all these alternatives.

5.1. Rendering using Static Rules

For the purposes of performance analysis, we consider that rendering based on static rules involves two processes:

1. compute the point index of a point $\boldsymbol{p}$ that lies at depth D_p on a ray starting from the corresponding pixel. For static rules, this information is sufficient for directly computing the interior/exterior classification.
2. find the first fragment after the clipping depth that lies between two areas with different interior/exterior classification. This step corresponds to trimming, i.e. rendering only the trim.

We have realized this on the GPU using several algorithms. Below we describe these implementation options, their details and their advantages and restrictions.

Point Index and Interior/Exterior Classification. One may use **front to back depth peeling - F2B** [4] for peeling all front-facing and back-facing intersections of the ray with the SCS (see Figure 11).

To improve the efficiency of the F2B peeling, we may use **dual depth peeling** [1], an extension of depth peeling based on a min-max depth buffer which peels two layers at a time. Instead of depth peeling one layer per pass, we apply the dual depth peeling method for peeling a pair of a front facing and a back facing fragment in one geometry pass.

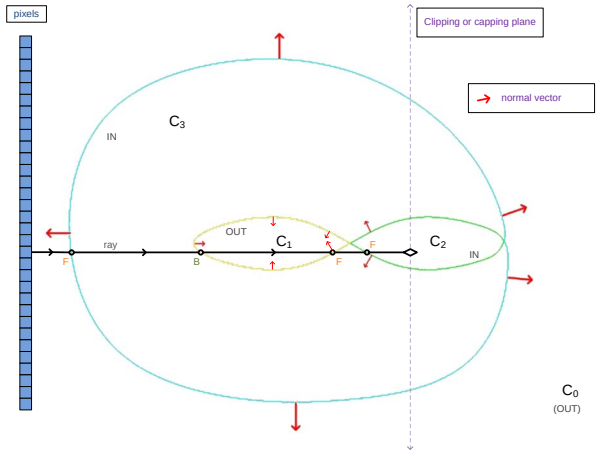


Figure 11: A self intersecting orientable surface that partitions space in four components: C_0 is the outside component, components C_2 and C_3 are interior and C_1 is exterior according to the alternating border rule. The boundary surfaces of C_2 and C_1 are shown with green and yellow respectively, whereas the remaining boundary (part of the boundary of C_3) is illustrated in cyan. The green part of the boundary should be trimmed off according to the alternating border rule.

Inspired by sort-independent methods [17, 1] for approximating efficiently transparency rendering effects, we introduce a technique that determines the classification in two passes. The main advantage of this technique is that it does not require sorting of the individual fragment layers of the model. At the first pass, we compute the point index by turning off the hardware depth test and initializing the point index *indexP* to zero (in this algorithm there is no need for lock flags). Then, for each layer whose depth is less than D_p, we set its index count to 1 or -1 depending on whether it is front or back facing, respectively. Using the atomic ADD blending operation, we accumulate the final point index result. Then, by using a full screen pass we compute the interior/exterior classification applying the static rule on the point index. A second geometry pass is needed to retrieve the next layer after the target depth D_p. Overall, the two pass technique works in two steps:

1. *IndexClassificationStatic* - (Algorithm 1)
2. *ClosestRender* - (Algorithm 2)

Algorithm 1 IndexClassificationStatic()

/* compute point index using ADD blending on *indexP* */

1: $indexP \leftarrow 0$
2: **for each fragment** $f : depth(f) \leq D_p$ **do**
3: $\quad indexP \leftarrow (f$ is *front facing*) ? 1 : -1
4: **end for**

/* classify the point as interior or exterior by applying the static rule on the point index */

1: $charP \leftarrow static\_rule(indexP)$

Algorithm 2 ClosestRender()

/* find the closest fragment after the D_p using Z-Test */

1: **for each fragment** f **do**
2: $\quad$ **if** $depth(f) \leq D_p$ **then**
3: $\quad\quad discard(f)$
4: $\quad$ **else**
5: $\quad\quad color \leftarrow color(f)$
6: $\quad$ **end if**
7: **end for**

The same principle can be implemented by **buffer-based peeling** using the FreePipe [11] or Linked-Lists [30] technique by processing all stored fragments with $depth(f) \leq D_p$ and adjusting accordingly the signed sum of front and back facing fragments. This is accomplished by performing a depth based presorting of all fragments per pixel. In the case of FreePipe where we have a preallocated space with all fragment information per pixel, the sorting step can be carried out by further exploiting parallelization among the pixels.

All techniques are free from any read-modify-write (RMW) hazards. The F2B and dual depth peeling methods do not suffer from the z-fighting effect for fragments that have different normal direction (i.e. one front facing and one back facing fragment with the same depth). However, the algorithm does suffer from z-fighting fragments with the same normal direction (i.e. two front facing fragments or two back facing fragments may cause miscalculation of the point indices). The Two-pass, the FreePipe and the Linked-Lists techniques do not suffer from any type of z-fighting.

Clipping and Trimming with Static Rules. For rendering the clipped STS, we process fragments after the clipping plane until we find a fragment that has alternating interior exterior characterization on its two adjacent sides (interior/exterior or exterior/interior). This is the fragment that we shall render. To determine the classification of the point index of the corresponding point of the clipping plane $\boldsymbol{C}$, we use Algorithm 1 with respect to the depth of $\boldsymbol{C}$ (i.e. $D_p = depth(C)$).

5.2. Rendering using Dynamic Rules

The point index computation is the same as in the case of static rules. The interior classification for a certain point is a slightly more complicated process. The trimming process for dynamic rules is considerably more complex and is outlined below.

Interior/Exterior Classification. In clipping with dynamic rules, we need to pass over to the next animation frame the characterization (interior/exterior) and the index of the corresponding clipping plane point. For this reason, we use one more texture with the index of the previous frame and the interior/exterior characterization. In fact, we may use the index and the interior exterior characterization of the point of the clipping plane that corresponds to any of the previous frames of the current sequence of disjoint deformations. For efficiency, we use the first frame of each such sequence (also called the *reference frame*. The corresponding texture information (index and interior/exterior classification) will be used in all frames of the current disjoint deformation sequence. The correctness of this process is due to the fact that during each sequence of disjoint transformations the index of each point may be altered only once. For this process, we use a variation of the *static* methods. The details are shown in Algorithm 3.

Clipping and Trimming with Dynamic Rules. For rendering the STS with dynamic rules, we need to have available all previous interior/exterior characterizations and index information from the reference frame. To do this, we need all interior/exterior information, the location of all fragments (i.e. the corresponding depths), and information to compute the corresponding indices. It is prohibitive to maintain all this information per pixel and pass it over from one shader to the next. A test implementation demonstrated that this is feasible using all available texture memory for 128 layers of fragments but this would also speed down considerably the rendering algorithm and would require powerful state of the art graphics hardware.

We present an algorithm that maintains the geometry information of the reference frame and uses coding for distinguishing whether a fragment has been derived from a primitive of: (i) the reference frame, (ii) the current frame or (iii) both (has not been deformed).

This information is compiled through the geometry shaders using the value of the *frameClass* parameter to store the frame classification of the primitive. This is then conveyed to the corresponding fragments. If $frameClass = 0$ then this fragment has originated from a primitive that is part of the reference frame only. If $frameClass = 1$ then the fragment has originated from a primitive that is part of the current frame only. Finally, $frameClass = 2$ means that the fragment has originated from a primitive that exists in both the current and the reference frames.

Thus, for each pixel we have available all fragment information from the current frame and the reference frame including depth information. We can also compute the corresponding indices for each such fragment of the current or the reference frame. In addition to this information we need a bit vector that will store the in/out information per pixel that corresponds to the characterization of the partitioning of space by the corresponding fragments. This needs to be computed only once for each reference frame. The size used to store this information sets a bound for the number of layers that we can peel. If we use a 4×32 bit vector we can account for 128 fragment layers

per frame, a trade off that is quit reasonable even for commodity graphics hardware.

The algorithm at a high level uses the following registers per pixel that are passed on to the next step: the current depth of the fragment we are processing *cDepthP* (initialized to the depth of the clipping plane $\boldsymbol{C}$); the final color of the first fragment that will not be trimmed: *color* (initialized to 0, this variable is also used as a lock flag); the *cIndexP* that is the index initially at the clipping plane and then after each processed fragment at the current frame; the *rIndexP* that is the index in the context of the reference frame initially at the clipping plane and then after each processed fragment; the bit vector *rPC* (reference frame partitioning characterization vector) that stores the interior/exterior characterization of the areas between fragments of the reference frame (computed for each reference frame only); the *rCount* is the fragment index for the fragment classification bit vector of the reference frame and the local registers CL_b, CL_a that maintain the characterization of the points before and after the fragments in the current frame. The algorithm is carried out in two steps:

1. *IndexClassificationDynamic* - (Algorithm 3)
2. *TrimRenderDynamic* - (Algorithm 4)

IndexClassificationDynamic() is invoked only once and determines the indices of the clipping plane at the reference and the current frames.

Algorithm 3 IndexClassificationDynamic

/* compute indices using ADD blending on [*cIndexP*, *rIndexP*] */
1: $[cIndexP, rIndexP] \leftarrow [0, 0]$
2: **for each fragment** $f : depth(f) \leq D_p$ **do**
3: $w_f \leftarrow (f$ is *front facing*$)\ ?\ 1\ :\ -1$
4: $cIndexP \leftarrow (frameClass\ >\ 0)\ ?\ w_f\ :\ 0$
5: $rIndexP \leftarrow (frameClass\ \neq\ 1)\ ?\ w_f\ :\ 0$
6: **end for**

/* classify the point as interior or exterior by applying the dynamic rule on the point index */
1: $cCharP \leftarrow dynamic\_rule(cIndexP, rIndexP, rCharP)$
2: **if** *current frame* == *reference frame* **then**
3: $rCharP \leftarrow cCharP$
4: $rIndexP \leftarrow cIndexP$
5: **end if**

During *TrimRenderDynamic*, we process one layer at a time, until we find the first fragment that should be rendered. Each step of the *TrimRenderDynamic* algorithm corresponds to either a separate shader invocation (multi-pass peeling) or processing the next fragment in the sorted fragment list (FreePipe or Linked-Lists).

5.3. Capping and CSG Operations

We have implemented **capping** at no extra cost by subtracting a capping box from our object (see CSG operations below) or by simulating the result of a front facing plane that has extended the exterior towards the capping plane (see Section 4). This will clip and cap the target object. We can render this using the algorithms described in the previous sections by setting the clipping plane outside the object since no additional clipping needs to be performed in this case.

Algorithm 4 TrimRenderDynamic

/* continue until we find the first non trimmed boundary fragment */
1: $cDepthP \leftarrow depth(C)$
2: **while** $color \neq 0$ **do**
3: **obtain the next fragment** f: $depth(f) > cDepthP$
4: $cDepthP \leftarrow depth(f)$
5: $w_f \leftarrow (f$ is *front facing*$)\ ?\ 1\ :\ -1$
6: **if** $frameClass \neq 1$ **then**
7: $rCharP \leftarrow rBC[++rCount]$
8: $rIndexP \leftarrow rIndexP + w_f$
9: **end if**
10: **if** $frameClass > 0$ **then**
11: $cIndexP \leftarrow cIndexP + w_f$
12: $CL_b \leftarrow cCharP$
13: $CL_a \leftarrow dynamic\_rule(cIndexP, rIndexP, rCharP)$
14: $cCharP \leftarrow CL_a$
15: **if** $CL_a \neq CL_b$ **then**
16: $color \leftarrow color(f)$
17: **end if**
18: **end if**
19: **end while**

Constructive solid geometry can be supported, allowing a modeler to create a complex surface by using Boolean operators even between complicated self-crossing objects. If rBC_A and rBC_B are the reference partitioning characterization bit vectors of manifolds *A* and *B* respectively, then *union*, *intersection* and *difference* can be implemented by simply performing bitwise operations on these vectors to compute the current point characterization and fragment trimming of the resulting CSG object. In this case, index and dynamic rule computations for each fragment of the current or the reference frame are not needed, thus speeding up the trimming of the CSG operation result.

The major drawback of the multi-pass version of the algorithms is that the fragment layers with depth equal to the associated depth from the previous pass are discarded and so not peeled. We have employed the technique of [27] that correctly resolves this limitation by using one more extra geometry pass in the expense of performance.

6. Implementation and Results

To demonstrate our technique, we have applied the rendering algorithms to a user controllable animation setting using both static and dynamic rules. We have used two types of concurrent local deformation operations: local influence deformations in conjunction with Laplacian smoothing and control point movement of NURB surfaces combined with mesh subdivision. The animation starts with an orientable self-crossing manifold whose areas are classified and trimmed using any static rule. At each step the user applies a sequence of concurrent disjoint deformations. The user may select different rules for each set of concurrent deformations. We have implemented a GPU-based linear space warp deformation tool to offer users the capability of creating arbitrary animation sequences. This tool works the

same way a magnet does; it attracts many vertices at each frame forcing the surface to deform smoothly. Vertices other than the selected vertices are affected within a geodesic distance range. An angle-weighted estimation process computes the attenuated deformation at each vertex. When working with non-dense geometry, it can become difficult to apply extreme stretches to the vertices without causing nasty lumps and creases on the model surface. To correct this effect, we have implemented iterative Laplacian smoothing and weighted vertex normal recomputation as separate GPU steps performed after the deformation process. Alternatively, for Bspline or NURB-based meshes the user may edit the control polygon and adjust the subdivision to derive a set of concurrent deformations.

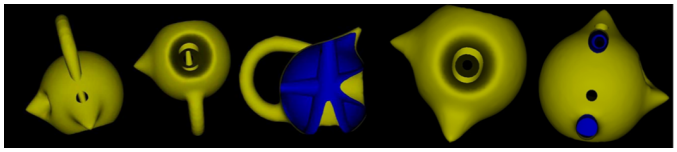


Figure 12: Rendering of the trim using clipping after applying several sets of deformations.

We have implemented interior/exterior point classification and rendering using the algorithms presented in Section 5. The implementation is based on OpenGL 4.2 using framebuffer objects with high precision 8bit integer, 32bit floating internal texture formats and 32bit floating point depth buffer precision.

All experiments were carried out on a commodity desktop with Intel Core i7-870 2.93GHz, 4GB DDR3 memory and NVIDIA GeForce GTX 480 graphics hardware. The visual results demonstrated by the figures and the videos use resolution of 1024×768 pixels which yields reasonably high quality results. We have implemented classification, trimming and rendering in conjunction with clipping and capping.

Figure 12 illustrates the result of rendering a self-trimmed surface that has undergone a series of deformations. In this case, the result is the same using either the static alternating border rule or the dynamic confluent deformation rule. Yellow visualizes front facing geometry whereas blue visualizes back facing geometry. The snapshot in the center uses clipping for inspecting the interior of this complex self-crossing manifold. Figure 14 illustrates the result of applying several deformations on the NURB surface (a) by moving the control points. The unintuitive hole and bump that are introduced when rendering the trim using the static rule (d,f), are eliminated through the use of the dynamic rule (e,g). The unintuitive hole is created due to the fact that the extended positive surface meets a previously negative area that is not part of the trim before the deformation. The unintuitive bump is created due to the fact that the extruded hole meets a previously internal highly positive index area that is not part of the trim before the deformation. For a better visual understanding of the unintuitive holes the reader is referred to the supplementary material (see also `http://www.cs.uoi.gr/~fudos/trimming.html`). Figure 15 illustrates the result of using capping (c) for inspecting the interior of model (a). Here we use a striped texture for rendering the capped part. Figure 16 illustrates CSG operations between two

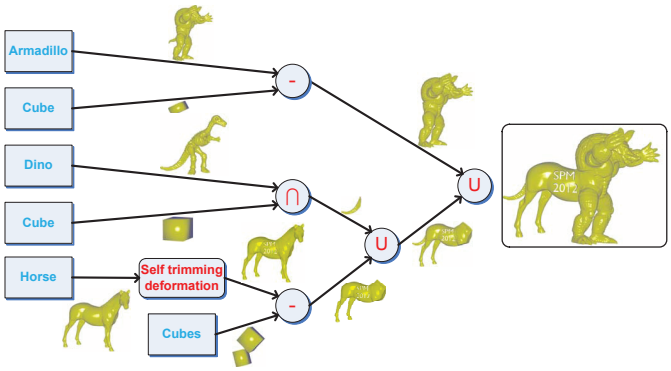


Figure 13: Rendering a deformed horse model combined with several other complex object parts with CSG operations.

copies of a highly deformed sphere. Finally, Figure 13 demonstrates the combination of deformed horse (where the original surface has been pushed through the object to create the SPM logo) with several other objects to render a horse-armadillo with dinosaur tail. The resulting object has around 500k faces producing 2 million fragments and is being rendered at 20-100 fps depending on the method used. More visual results are presented in the supplementary material (videos) of this paper.

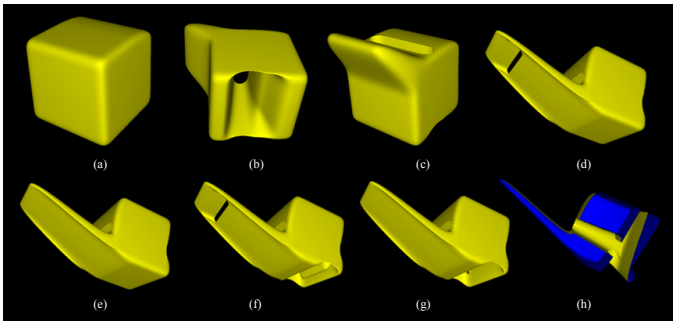


Figure 14: (a) The original NURB surface. (b and c) After applying a set of disjoint deformations. (d) Rendering the result of one more deformation with the static rule, where the upper extrusion is deformed so as to cross an extended hole (observe the unintuitive hole in the upper part) and (e) the same using the dynamic rule (no hole is present in the upper part). (f) Rendering the trim with the static rule after one more hole is created at the bottom, observe the unintuitive bump at the bottom and (g) the same with the dynamic rule (no bump is present). (h) g with clipping.

Table 2 presents a comparative performance evaluation of the proposed algorithms for in/out classification and index computation without trimming for models with different characteristics. For all peeling methods, we provide the resulting frames per second and the number of passes needed for point classification. Two pass performs very well as compared to all other methods including the Linked-Lists and the FreePipe. We observe that the linked lists approach that creates linked lists of all fragments per pixel yields poor results in terms of efficiency because of extensive memory access contention. Thus, the linked list implementation depends a lot on the number of the gener-

Table 2: FPS, number of passes and memory needed using the static rule without trimming.

Resolution 1024x768				Static Rules																				Dynamic			Standard	
Model				F2B							Dual							2 passes			FreePipe			Linked Lists			2 passes			without	
Name	Size			Best		Average		Worst			Best		Average		Worst			All			All			All			All			All	
	Vertices	Faces	Fragments	Fps	Passes	Fps	Passes	Fps	Passes	MB	Fps	Passes	Fps	Passes	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes
Homer	5103	10202	0.991M	510		212	7	121	16		490		203	5	113	9		940			575		90	111		20,34	895			1355	
Sphere	7478	14952	0.897M	500		263	6	180	10		465		240	4	162	6		1055			620		54	123		19,26	990			1518	
Deformed sphere	7478	14952	0.933M	515	2	219	7	135	14	21,75	485	1	194	5	125	8	31,5	965	2	7,5	608	1	78	115	1	19,67	905	2	9,75	1465	1
Deformed Nurbs	23807	47557	2.57M	330		149	7	88	14		300		120	5	76	8		520			312		78	47		38,4	494			824	
Armadillo	172974	345944	0.8M	213		51	9	31	16		245		73	6	51	9		273			350		90	30		18,2	265			461	
Dragon	437645	871414	1.3M	121		23	12	12	22		162		38	7	22	12		198			258		126	38		23,83	196			286	

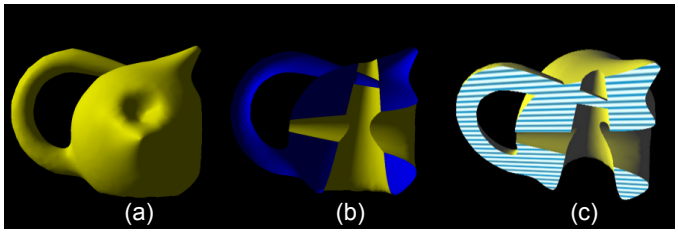


Figure 15: Rendering the deformed self-trimmed surface of (a) using (b) clipping and (c) capping.

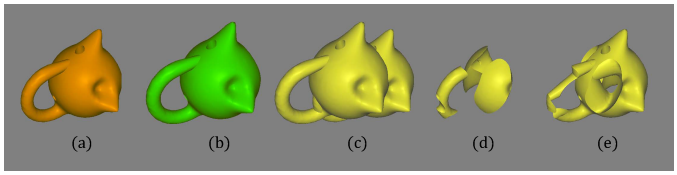


Figure 16: (a) Object A. (b) Object B. (c) Rendering $A \cup B$, (d) $A \cap B$ and (e) $B - A$.

ated fragments. The FreePipe based technique achieves reasonably good performance in the expense of increased memory requirements. Moreover, FreePipe scales better that Linked-Lists with respect to resolution increase.

Table 3 presents performance results for rendering the trim using our technique with (a) multi-pass depth peeling (b) FreePipe peeling and (c) peeling using Linked Lists on a NURB surface with three tessellation levels, a deformed self-crossing sphere and several non self-crossing models.

In the *Homer, Armadillo* and *Dragon* models, static depth peeling stops after extracting the first layer (using two geometry passes) since these are non self-crossing. On the other hand, the multi-pass algorithm on self-trimmed models depends on their maximum depth complexity since peeling is needed until all pixels find a fragment with alternating in/out classification on its two sides. For the NURBS and the deformed sphere three to five rendering passes are required. For the dynamic case, we choose the worst case scenario using as current and reference model the same model, leading at poor performance due to the large number of layers processed. The FreePipe technique outperforms static multi-pass peeling since only one geometry pass is needed to store the entire fragment array. Furthermore, the overhead for the dynamic trimming is considerably smaller (around 50%) as compared to the overhead for the multi-pass peeling (> 60%). Finally, the limitations of the Linked-Lists algorithm discussed in Table 2, hold for the trimming version as well.

7. Conclusions and Future Work

We have explored semantics for classifying the interior of self-crossing manifolds and have introduced algorithms for efficiently rendering the resulting trimmed boundary on the GPU. This scheme can be used for rendering deformation animations of self-crossing surfaces. Another well fitted application is to preview the interior of solids using capping prior to performing free form editing. The fast interior and boundary detection and rendering have been extended to CSG originated results among two or more self-crossing manifolds.

Fast collision detection in complex scenes may be realized using the techniques presented in this paper. The algorithms in this paper may be utilized for voxelizing the interior and then produce an actual mesh for the STS. The semantics of the dynamic rules may be extended to capture non dijoint concurrent deformations.

References

[1] Louis Bavoil and Kevin Myers. Order Independent Transparency with Dual Depth Peeling, 2008.

[2] David Eppstein and Elena Mumford. Self-overlapping curves revisited. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 160–169, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[3] D. Epstein, F. Jansen, and Rossignac J. Z-buffer rendering from csg: The trickle algorithm. *IBM Research Report*, RC15182, 1989.

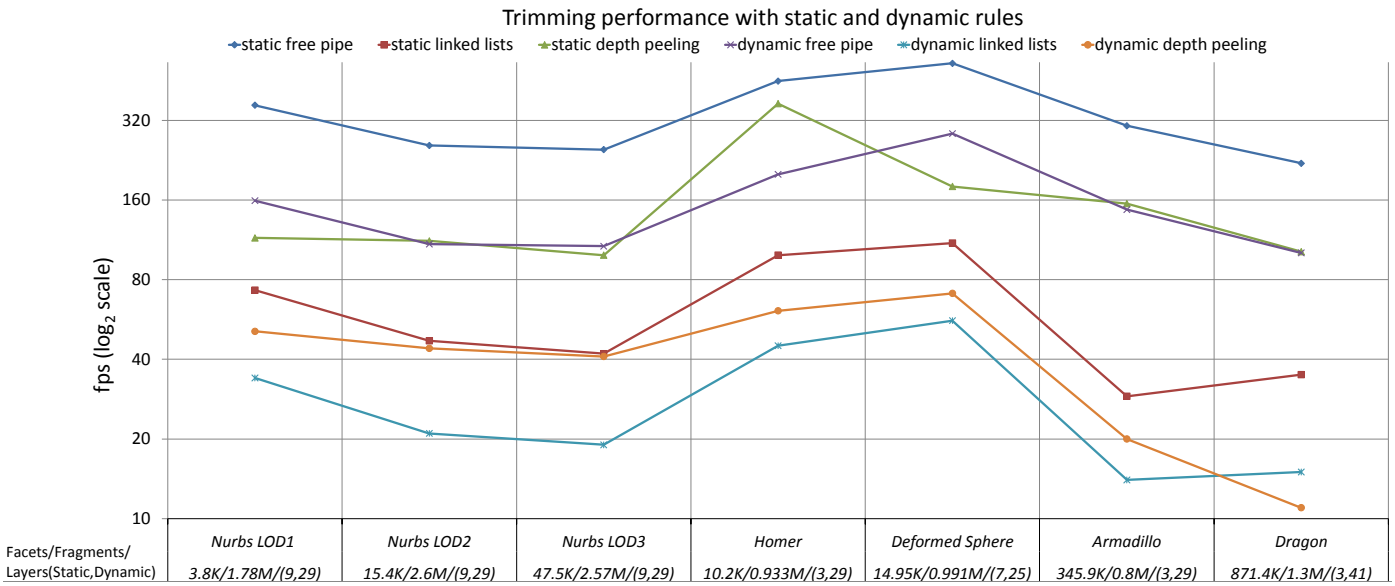
[4] Cass Everitt. Interactive Order-Independent Transparency, Tech. Report, Nvidia Corporation, 2001.

[5] Sudipto Guha, Shankar Krishnan, Kamesh Munagala, and Suresh Venkatasubramanian. Application of the two-sided depth test to CSG rendering. *Proceedings of the 2003 symposium on Interactive 3D graphics - SI3D '03*, page 177, 2003.

[6] John Hable and Jarek Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph.*, 24:1024–1031, July 2005.

[7] John Hable and Jarek Rossignac. CST: constructive solid trimming for rendering BReps and CSG. *IEEE transactions on visualization and computer graphics*, 13(5):1004–14, 2007.

Table 3: Performance results for rendering a self-trimmed surface using static and dynamic rules.



[8] Jeff Alan Heisserman. *Generative geometric design and boundary solid grammars*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-16022.

[9] Frederik W. Jansen. Depth-order point classification techniques for csg display algorithms. *ACM Trans. Graph.*, 10:40–70, January 1991.

[10] Anil Kaul and Jarek Rossignac. Solid-interpolating deformations: Construction and animation of pips. *Computers & Graphics*, 16(1):107–115, 1992.

[11] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Efficient depth peeling via bucket sort. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, page 51, 2009.

[12] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM.

[13] Ignacio Llamas, Byungmoon Kim, Joshua Gargus, Jarek Rossignac, and Chris D. Shaw. Twister: a space-warp operator for the two-handed editing of 3d shapes. *ACM Trans. Graph.*, 22(3):663–668, July 2003.

[14] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9:43–55, July 1989.

[15] Marilena Maule, Joo L.D. Comba, Rafael P. Torchelsen, and Rui Bastos. A survey of raster-based transparency techniques. *Computers & Graphics*, 35(6):1023 – 1034, 2011.

[16] James McCann and Nancy Pollard. Local layering. *ACM Trans. Graph.*, 28:84:1–84:7, July 2009.

[17] Houman Meshkin. Sort-independent alpha blending. In *Game Developers Conference 2007*, 2007.

[18] Uddipan Mukherjee, M. Gopi, and Jarek Rossignac. Immersion and embedding of self-crossing loops. In *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling*, SBIM '11, pages 31–38, New York, NY, USA, 2011. ACM.

[19] T. M. Murali and Thomas A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 155–ff., New York, NY, USA, 1997. ACM.

[20] Aristides A.G. Requicha and Herbert B. Voelcker. Boolean operations in solid modelling: Boundary evaluation and merging algorithms. 1984.

[21] J. Rossignac. Ordered boolean list (obl): Reducing the footprint for evaluating boolean expressions. *Visualization and Computer Graphics, IEEE Transactions on*, 17(9):1337 –1351, sept. 2011.

[22] J R Rossignac and A A G Requicha. Offsetting operations in solid modelling. *Comput. Aided Geom. Des.*, 3:129–148, August 1986.

[23] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: cross-sections and interferences. *SIGGRAPH Comput. Graph.*, 26:353–360, July 1992.

[24] Jarek R. Rossignac and Maarten van Emmerik. Hidden contours on a frame-buffer . In P F Lister, editor, *Eurographics Workshop on Graphics Hardware*, pages 188–203, Cambridge, UK, 1992. Eurographics Association.

[25] Jaroslaw R. Rossignac and Herbert B. Voelcker. Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Trans. Graph.*, 8:51–87, November 1988.

[26] R.B. Tilove and A.A.G. Requicha. Closure of boolean operations on geometric entities. *Computer-Aided Design*, 12(5):219 – 220, 1980.

[27] Andreas Vasilakis and Ioannis Fudos. Z-fighting aware Depth Peeling. *SIGGRAPH '11: ACM SIGGRAPH 2011 Posters*, 2011.

[28] H. Voelcker, A. Requicha, E. Hartquist, W. Fisher, J. Metzger, R. Tilove, N. Birrell, W. Hunt, G. Armstrong, T. Check, R. Moote, and J. McSweeney. The padl-1.0/2 system for defining and displaying solid objects. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 257–263, New York, NY, USA, 1978. ACM.

[29] Yajun Wang, Jiaping Wang, Nicolas Holzschuch, Kartic Subr, Jun-Hai Yong, and Baining Guo. Real-time rendering of heterogeneous translucent objects with arbitrary shapes. *Computer Graphics Forum*, 29(2):497–506, 2010.

[30] Jason C. Yang, Justin Hensley, Holger Grn, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.

[31] Hanli Zhao, Charlie Wang, Yong Chen, and Xiaogang Jin. Parallel and efficient boolean on polygonal solids. *The Visual Computer*, 27:507–517, 2011. 10.1007/s00371-011-0571-1.