Eurographics Symposium on Geometry Processing 2009
Marc Alexa and Michael Kazhdan
(Guest Editors)

Volume 28 (2009), Number 5

Fast, Exact, Linear Booleans

Gilbert Bernstein[1] and Don Fussell[1]

[1]University of Texas at Austin, United States

Abstract

We present a new system for robustly performing Boolean operations on linear, 3D polyhedra. Our system is exact, meaning that all internal numeric predicates are exactly decided in the sense of exact geometric computation. Our BSP-tree based system is 16-28$\times$ *faster at performing iterative computations than CGAL's Nef Polyhedra based system, the current best practice in robust Boolean operations, while being only twice as slow as the non-robust modeler Maya. Meanwhile, we achieve a much smaller substrate of geometric subroutines than previous work, comprised of only* 4 *predicates, a convex polygon constructor, and a convex polygon splitting routine. The use of a BSP-tree based Boolean algorithm atop this substrate allows us to explicitly handle all geometric degeneracies without treating a large number of cases.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computing Methodologies—Computational Geometry and Object Modeling;

1. Introduction

Despite a long history of robustness issues, Boolean set operations (aka. constructive solid geometry or CSG for short) remain a popular choice in most 3d modeling systems available. The operation conforms nicely to architectural conceptions of positive and negative space in building massing, the machining of mechanical parts in CAD/CAM, and general purpose "sculpting" operations of adding and removing virtual stuff from a shape, to mention a few applications. We built the system described here to address such sculpting applications, for which we found existing Boolean operations lacking in speed and robustness, particularly when performing iterated operations, such as repeated gouging of an object with a tool. Sculpting is a particularly challenging application, since even moderate success requires fast(real-time) and completely robust(zero failure) solutions in order to be usable. We demonstrate that Boolean operations on polyhedra can be made fast and robust enough to be a viable component of such applications, contrary to the expectations of many in the computer graphics community.

One of the first and biggest motivations for research into robust geometry was the not uncommon experience of seeing a system crash in response to attempting to perform Boolean operations. However, despite 20 years of research on the problem, state of the art robust Boolean operations [HK05] may run up to 20 times slower than non-robust operations. In our experiments(§4), these performance characterizations are actually optimistic. We see cases where the state of the art (CGAL) takes more than 50 times as long as non-robust Booleans. When we spoke to practitioners who implement Boolean operations in entertainment and architectural design modelers, none were aware of any research into geometric robustness. They believed that there are no solutions to their robustness problems short of using Matlab/Mathematica-style big number computation, which they deem prohibitively expensive. Given that the aforementioned Hachenberger et al. paper [HK05] constitutes the only experiment with more than a single test in the last 20 years, it is easy to see how this impression persists.

Contribution We demonstrate a system capable of computing completely robust (i.e. exact §2.1) linear Boolean operations only twice as slowly as the non-robust (i.e. fragile §2.1) modeler Maya, providing the first empirical evidence that geometric robustness techniques are actually practical for computing Boolean operations. This constitutes the second and largest comparative test between non-robust commercial Booleans and robust Boolean systems ever conducted and published as far as we are aware. We accomplish this using a novel synthesis(§3) of techniques mostly $15-20$ years old(§2). In addition to a number of small but important changes to these techniques, we present a new plane-based convex polygon splitting algorithm(§3.2).

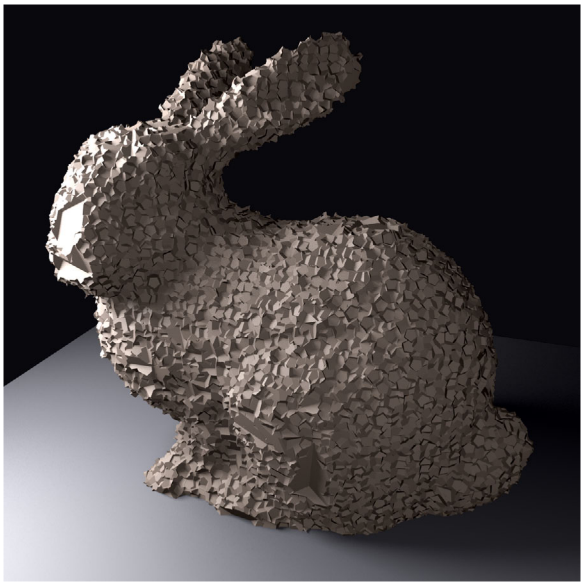


Figure 1: *This bunny was sculpted out of a block using* ~ 9000 *subtractions of dodecahedra. Our system computed these subtractions in approximately* 3 *hours at a rate of about* 1*Hz. Using extrapolations from smaller experiments, this computation would have taken CGAL at least* 3 *days. Maya isn't even able to perform this computation, crashing within the first couple hundred operations.*

We describe input/output methods(§3.4) for conversion between point-based polygonal meshes and plane-based binary space partitioning trees, but I/O is not our focus. We make no strong guarantees about the accuracy of our conversions(e.g. topology preservation), but can perform the conversion arithmetic accurately to within machine epsilon. Furthermore the current system performs robustly (no crashes/failures) for both plane-based and point-based inputs, additionally ensuring 100% accurate results for inputs already represented by planes, such as when sculpting with primitive polyhedra—our original motivation.

2. Background

2.1. Robustness

Following Yap [Yap04], "nonrobustness refers to qualitative or catastrophic failures in geometric algorithms arising from numerical errors." That is, geometric robustness is not the same as precise numerics. Numeric precision for its own sake is usually irrelevant, so long as the results aren't grossly inaccurate. However, small numeric errors will sometimes lead to catastrophic program failures (inconsistencies) or grossly incorrect results. Geometric robustness seeks to avoid these byproducts of numeric errors without incurring the cost of fully precise numerics. Thus, geometric robustness is and has always been about simultaneous speed and robustness. Either goal is easy to achieve in the absence of the other.

Numeric computations in a geometric program may be classified as either **predicates** or **constructions** [She06]. Predicates make two-way or three-way decisions based on numeric coordinates of geometric objects, while constructions compute the coordinates for new geometric objects from the known coordinates of existing geometric objects. In general, algorithms that restrict themselves only to predicates are easier to make robust and fast than algorithms which use constructions. By restricting to predicates, each of which makes a decision based on a few numbers, the depth of any arithmetic expression has an *a priori* constant bound. This bound allows for static filters [She97, FVW96] that produce fast predicates that are exact, meaning they always give the correct branching answer. By way of contrast, algorithms that use constructions allow for the construction of arbitrary depth arithmetical expressions, incurring significant speed penalties to guarantee exact answers. In this work, we make use of Sugihara and Iri's observation [SI89] that linear Booleans require no constructions if representations are based on planes and not points.

Our solution is **exact** in that it is unconditionally robust given consistent input. It is also a **fixed-precision** technique since we can, by the use of plane-based representations, avoid constructions and thus provide an *a priori* constant bound on the depth of any arithmetic trees used and thus on the representational complexity required. This is in contrast to **arbitrary-precision** techniques which have no such bound, generally because they use constructions [BMP93, GHH*03]. As we will demonstrate (§4) fixed-precision techniques are much faster than arbitrary-precision techniques. Devillers and Pion [DP03] provide corroborating evidence in their Delaunay triangulation experiments.

There are a variety of non-exact approaches that provide weaker notions of robustness. Well-known examples include epsilon-tweaking [NAT90, LTH86], in which heuristic tolerance parameters are adjusted in hope of successful execution. Industrial systems for CAD/CAM and solid modeling generally employ this technique along with undo operations to allow the user to work around operations that fail, but it is a **fragile** system in that it provides no guarantee against failure of an operation. Interval geometry techniques [Seg90, Bru91], which represent geometric primitives with local tolerances within guaranteed intervals, are more principled. They are **quasi-robust** in that they provide guaranteed bounds on the imprecision of the representation, but these intervals can become arbitrarily wide, eventually collapsing all the geometry to a point within a huge interval. As with epsilon-tweaking, systems based on this approach can be quite useful if users are willing to work around these problems. However an efficient exact system is preferable (and sometimes necessary) in order to avoid user workarounds.

General Position and Geometric Degeneracies

In addition to the aforementioned numeric issues, many algorithms rely on "general position" assumptions, such as, "In general, two surfaces (2d) intersect in some number of curves (1d).". Local arrangements of geometry that violate these assumptions are known as (geometric) degeneracies, or degenerate arrangements. One approach to geometric degeneracies is to not make general position assumptions—we do this. However, for some algorithms this approach leads to a large, unwieldy explosion of degenerate cases. Thus geometry is often perturbed into general position. Done naively, (by directly modifying coordinates) perturbations produce further complications. Therefore, robust perturbation [Sei94] is symbolic, further necessitating the use of exactness.

2.2. Booleans

We (along with Maya and industry) follow Requicha's regularized sets formalism [Req77]. CGAL follows the Nef formalization [Nef78].

There are many specific approaches to handling Booleans in the literature, but since we have to interoperate with surface modelers, we are concerned with those which support boundary evaluation and thus can output meshes. **B-rep** algorithms [HHK89, LTH86] are the most common of these, and they share an algorithmic template with the more recent cell complex algorithms [GHH*03, HK05] . This template is: 1) If A and B are the boundaries of two objects whose union, difference or intersection we would like to compute, find the intersection of A and B, thus dividing each surface into two components, one inside and one outside the other surface. 2) Select the appropriate component of each surface, and 3) stitch these together to form the correct output. This apparent simplicity belies the large number of special cases that result from the various ways the two objects can align [Hof89]. Resulting systems can be quite complex, making it difficult to ascertain whether all cases have been handled. This case analysis can be reduced by the use of perturbation techniques(§2.1) [For97], but we have yet to come across any publication that explicitly presents a complete geometric substrate for a B-rep algorithm.

CSG trees encode objects as a Boolean combination of primitive objects (e.g. $(A \cup (B \cap C)) - D$). Computing Boolean operations on CSG trees is often straightforward, the real problem is producing the resulting boundary [RV85]. This can be accomplished by reduction to B-rep algorithms or by algorithms that operate globally on the tree and exploit properties of a restricted set of primitives [Bru91, BR96]. However, these methods are inappropriate for iterative, and thus interactive applications.

BSP trees afford an alternative to B-rep algorithms that avoid their concomitant case explosion by explicitly handling all degenerate configurations of geometry [TN87, NAT90]. These have been demonstrated to have suitable performance for interactive volumetric sculpting [Nay90], and recent work [LDS09] suggests further possible performance improvements. Unfortunately, all of these approaches are fragile and therefore subject to failures and user workarounds. We will detail this approach shortly(§3.3).

Aside from the BSP tree work, the closest precursor to our system is that of Fortune [For97]. Like us, Fortune employed both Sugihara and Iri's observation on planes and fixed-precision techniques for robustness. Beyond this, we have very little in common. We use different predicates, a completely different representation: BSP trees instead of B-reps, and we do not rely on symbolic perturbations, instead choosing to handle all degenerate arrangements explicitly (and with a simpler substrate). These differences are all the result of key design decisions toward our goal of combined robustness and performance.

Sampled volumes offer another alternative—gridded discretization—often advocated for sculpting contexts [GH91]. This approach was refined by Ferley, Cani and Gascuel [FCG00], and reached its zenith with Perry and Frisken's Kizamu [PF01], capable of handling up to 12-ply deep octrees. Two major shortcomings of Kizamu dissuaded us from pursuing volumetric approaches further: (a) rendering becomes prohibitively expensive as 12-play octrees are approached and (b) discretization errors, more deleterious to perceived visual quality than roundoff, are still present.

3. System Design

Although we began by trying to build a Boolean sculpting system based on Thibault and Naylor's BSP tree algorithms [TN87, NAT90], it is, perhaps, easier to motivate and explain our design in reverse; that is, bottom up. As Sugihara and Iri pointed out, one must use planes, not points as the fundamental primitive in order to make use of fixed precision techniques (§2.1). We proceed from this decision upwards. Naturally, our numeric predicates must all be phrased as predicates about planes. Likewise, the geometric definitions and subroutines must now bootstrap off of these predicates and thus be completely phrased in terms of plane manipulations. Finally, binary space partitions are employed because they are (a) plane-based, and (b) allow for economy in the geometric substrate.

3.1. Numeric Substrate

A plane, in our system, is taken to be a quadruple of floating point numbers, (a,b,c,d) interpreted as the coefficients of a plane equation. On occasion we will have cause to refer to "points", by which we do not mean coordinate triples. Rather, we take a point to be a triple of planes (p,q,r) whose mutual intersection is the point in question. We do not use or refer to lines. We provide exactly four arithmetical predicates concerning planes and their relative positioning in space [YN97, Sto87].

coincidence. Two planes p,q are coincident, if and only if the determinants of all 2×2 minors of the following matrix are zero:

$$\begin{bmatrix} p_a & p_b & p_c & p_d \\ q_a & q_b & q_c & q_d \end{bmatrix}$$

coincident orientation. Furthermore if p and q are known to be coincident, then p is similarly oriented to q if and only if all products p_aq_a, p_bq_b, p_cq_c, p_dq_d are non-negative.

point validity. A point (p,q,r) is well-defined (i.e., valid) if and only if the following determinant is non-zero:

$$\begin{vmatrix} p_a & p_b & p_c \\ q_a & q_b & q_c \\ r_a & r_b & r_c \end{vmatrix}$$

orientation. Given that the point (p,q,r) is valid, it lies behind, on, or in-front of the plane s if and only if the following expression is negative, zero, or positive, respectively:

$$\begin{vmatrix} p_a & p_b & p_c \\ q_a & q_b & q_c \\ r_a & r_b & r_c \end{vmatrix} * \begin{vmatrix} p_a & p_b & p_c & p_d \\ q_a & q_b & q_c & q_d \\ r_a & r_b & r_c & r_d \\ s_a & s_b & s_c & s_d \end{vmatrix}$$

These predicates are implemented as static filtered floating-point predicates in the style of Shewchuk [She97]. We make a few deviations from Shewchuk's pattern. First, our predicates only take single-precision floating point input, but perform computation using double precision. This allows us to guarantee the absence of exponent overflow/underflow, and allows us to tighten our static error bounds. Second, rather than using a cofactor expansion, we compute the 4×4 determinant as a dot product between two lines in Plücker coordinate form. This makes the arithmetic tree shallower, resulting again in tighter static error bounds. Finally, we only use single stage filters, for simplicity's sake.

3.2. Geometric Substrate

In the preceding section on the numeric substrate we defined planes as primitives, points as triples of planes and 4 predicates operating on these planes. We will now define a convex polygon type along with a constructor and splitting routine. These 2 subroutines constitute the entire geometric substrate. They are used to support sub-hyperplane construction, tree splitting, and input/output conversions(§3.3, §3.4).

convex polygon. We take a convex polygon (In this paper, 'polygon' always stands for 'convex polygon') to be a plane of support s along with a list of bounding planes $\{b_i\}_{i\in\mathbb{Z}_n}$ listed in counter clockwise order. The vertices of this polygon are then given by $v_i = (s, b_{i-1}, b_i)$.

Construction of a Convex Polygon from a Plane Given a plane h, this operation constructs a convex polygon representing h clipped by a "very large box." In this way, the output polygon serves as a stand in for the infinite extent plane. A consistent axis aligned "very large box" is used for all calls to the constructor and consists of the volume of space bounded by the planes $x^+, x^-, y^+, y^-, z^+, z^-$.

In order to form the resulting polygon the dominant component of h's normal is found. Without loss of generality, let it be h_z. Then a polygon P is formed combinatorially with h as support and x^+, x^-, y^+, y^- as bounds. The order of the bounds may be determined by inspecting the sign of h_z. Finally, P is clipped by z^+ and z^- using the (following) polygon splitting routine to ensure that the entire polygon lies within the "very large box."

Splitting a Convex Polygon by a Plane Since polygon splitting may be viewed as two successive and complementary instances of polygon clipping, we will present a polygon clipping algorithm for simplicity. Given a convex polygon $(s, \{b_i\}_{i\in\mathbb{Z}_n})$, and clipping plane h, we wish to output the part of the polygon lying to the positive side of h, or indicate that there is no such part.

Our algorithm is similar to Sutherland-Hodgman style polygon clippers, where one decides at each step "do or do not output the current vertex", except we decide "do or do not output the current bounding plane (edge)." Analogously, rather than deciding if and when to insert a crossing point into the output stream, we decide if and when to insert the splitting plane into the output stream. These changes are key to eliminating the construction of line-plane intersections, illustrating the "use planes, not points" principle.

```
If $s$ is coincident with $h$
  If $s$ is similarly oriented to $h$
    Return $(s, \{b_i\}_{i\in\mathbb{Z}_n})$
  Else
    Return nothing
Else
  For $i \in \mathbb{Z}_n$ (in order),
    Output planes as specified by table lookup using:
    $o(s, b_{i-2}, b_{i-1}, h)$
    $o(s, b_{i-1}, b_i, h)$
    $o(s, b_i, b_{i+1}, h)$
  Return $(s, \text{output})$
```

Algorithm: Clip $(s, \{b_i\}_{i\in\mathbb{Z}_n})$ by h

The algorithm proceeds by iterating through the polygon's bounding planes and deciding whether to output each bounding plane b_i. In order to make this decision the two endpoints of b_i, $v_i = (s, b_{i-1}, b_i)$ and $v_{i+1} = (s, b_i, b_{i+1})$ are tested against the clipping plane h. Additionally, the immediately preceding vertex, $v_{i-1} = (s, b_{i-2}, b_{i-1})$ is tested. As we proceed, the reason for testing this additional third vertex will manifest itself.

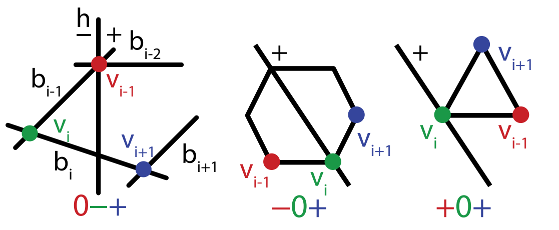


Figure 2: *(left) coding example (center & right) These two cases are indistinguishable using only two vertices.*

Given that each vertex may lie behind (−), on (0), or in-front (+) of the clipping plane h, each triple of vertices v_{i-1}, v_i, v_{i+1} may lie in one of 27 possible arrangements relative to the clipping plane h. As a shorthand we use the following coding scheme. Suppose that v_{i-1} lies on (0) h, that v_i lies behind (−) h, and that v_{i+1} lies in-front (+) of h. Then encode this arrangement with the sequence $0-+$. (Figure 2)

For each of these 27 codes, we must decide whether to output the current bounding plane b_i (signal B) or not (no signal); we must decide whether to output the clip plane h (signal H) or not (no signal); and if both are to be output, then we must decide on the relative order of output (BH or HB). To complete the procedure, the output planes (if there are any) are taken in order as the bounding plane list for the resulting clipped polygon.

In deriving the correct output, one should be careful of the following two pitfalls. One must make sure that the H signal is given no more than once, and one must not output both the bounding plane b_i and the clip plane h in the case that b_i's corresponding edge (v_i to v_{i+1}) lies in the clip plane h. We address the former pitfall by choosing to output h only when re-entering the positive halfspace. We address the latter pitfall by omitting the output of the bounding plane in question b_i, and by outputting the clip plane h as the polygon turns back into the positive halfspace.

Table 1: *Polygon Clipping Encodings (∗ is a wildcard)*

Input	Output	I	O	I	O
$*++$	B	$+0+$	B	$*-+$	HB
		$00+$	HB		
		$-0+$	HB		
$*+0$	B	$*00$	∅	$*-0$	∅
$*+-$	B	$*0-$	∅	$*--$	∅

Upon close inspection of this table, you can see that if the $+0+$ case output HB instead of B, then the entire table would reduce under symmetry to three or four cases, and appear even more similar to Sutherland and Hodgman style clippers. To understand why this asymmetry occurs, consider Figure 2, with two instances, one of a hexagon clipped along its diagonal and one of a triangle clipped at a single vertex. Using only two vertices, we do not have sufficient local information to distinguish whether or not the clipping plane should be output.

3.3. BSP Tree Set Operations

Algorithms for Boolean set operations on polyhedra using BSP trees were presented by Thibault, Amanatides and Naylor [TN87,NAT90]. This approach has the advantage over B-rep based approaches of handling degeneracies in a simpler and more uniform way, but it has been prone to numeric robustness issues. We use a modification of this approach that is completely plane-based so as to allow for use of our efficient, robust substrate. We present a high-level overview of the BSP tree approach, followed by a description of our deviations from the algorithms that are described in full detail in Thibault and Naylor [TN87] and Naylor et al. [NAT90].

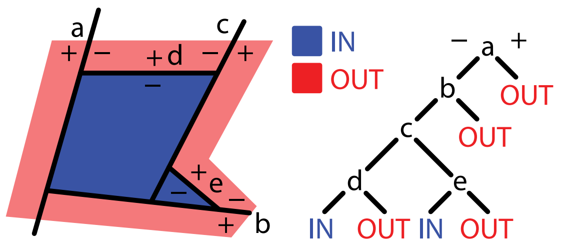


Figure 3: *BSP tree representing the blue object*

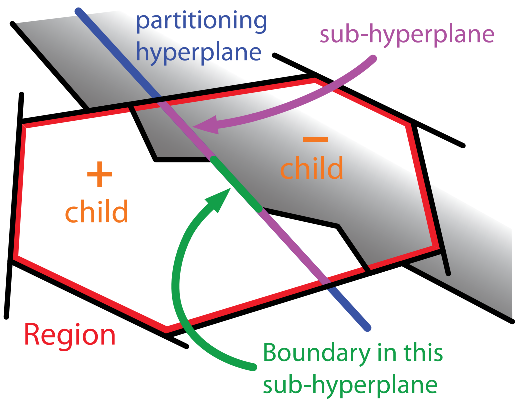


Figure 4: *parts of a BSP tree internal node (cell diagram)*

How does a BSP tree represent an object? Using planes, a BSP tree recursively partitions space into (convex, possibly unbounded) cells. The leaves of the BSP tree correspond to indivisible cells in the partition. By simply coloring each of the leaf cells with one of the two labels, IN or OUT, we get a representation for the polyhedral set consisting of all of the IN colored cells. (Figure 3) Internal nodes of the BSP tree represent divisible cells of the partition. Each such cell has an associated “region”, which is a set of halfspaces descended on a path to this node from the root. The intersection of these halfspaces defines this region, ie. the extent of the cell, geometrically. At the node of the tree, the partitioning (hyper)plane is stored, which along with the region allows for finding the sub-hyperplane, that part of the hyperplane lying in the node’s region. Additionally, positive and negative child pointers are maintained. (Figure 4)

How does a BSP tree represent the boundary of an object? The boundary is stored at the internal nodes of the BSP tree as an augmentation of the tree. Each internal node of the BSP tree stores the portion of the boundary contained within its associated sub-hyperplane, represented as a set of convex polygons. For us, these are plane-based polygons. For Thibault and Naylor, they were point-based polygons.

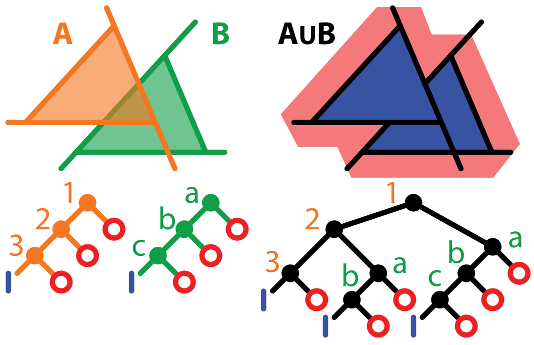


Figure 5: *result of a union operation*

How do you compute set operations, say $A \cup B$, between BSP trees? The two operand (binary space) partitions are “merged” into a new (binary space) partition (Fig 5). This new partition is found by repeatedly splitting one tree by the other’s root partition, thus reducing the set operation problem to two smaller instances at each step. The leaf cells of this final partition are either IN or OUT in each operand object A and B. Thus, the resulting leaf is colored IN if it is IN A or B and colored OUT if it is OUT of both A and B (and similarly with different logic for intersection and difference).

How are degeneracies dealt with? (i.e. How is the tree split?) Given a subtree T lying in a region (Fig 4), and a plane to split it with S, we split T’s sub-hyperplane ($T.shp$), and then recursively split T’s subtrees. In order to do so, we need only determine how $T.shp$ and $S \cap T.region$ lie with respect to each other. Figure 6 shows the 7 possible arrangements that arise. Determining which of these 7 hold is a simple matter of splitting $T.shp$ by S and $S \cap T.region$ by $T.hp$.

How do you compute the boundary of this new, resultant object $A \cup B$? The new boundary must be composed of some subset of the two old boundaries. Therefore, we can just classify the old boundaries, using the new tree, to see which faces we should keep around.

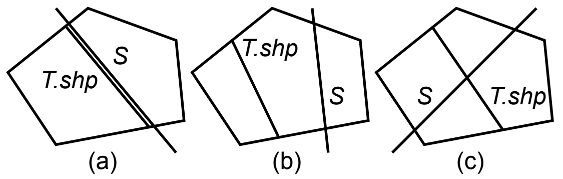
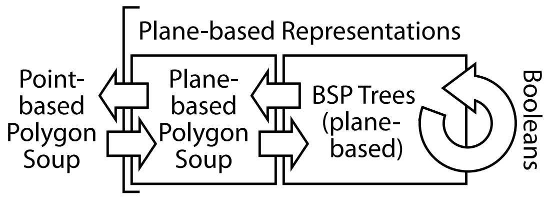


Figure 6: *(a) $T.shp$ and $S \cap T.region$ are coincident and either similarly or oppositely oriented. (b) $T.shp$ and $S \cap T.region$ do not intersect, and are relatively oriented in one of four possible combinations. (c) $T.shp$ and $S \cap T.region$ intersect.*

We can implement most of these operations recursively, over the BSP tree and its subtrees. Furthermore, we can be very efficient by interleaving the computation of the new boundary and BSP tree together. By using information gained in the midst of the computation, we can avoid unnecessary work, e.g. the recursive algorithm can quickly determine that a whole subtree of A lies disjoint from B, and therefore the boundary stored in this subtree is not modified by the set operation. In this way, BSP trees naturally exploit spatial locality by equating it with the structural locality of the tree structure.

Our particular implementation is a hybrid of techniques from Thibault and Naylor’s two papers [TN87, NAT90]. We take the “Merge” algorithm from the ’90 paper as our scaffold. Building onto that, we store the boundary as convex polygons at the internal nodes and maintain their consistency throughout the set operation, as advised. In a departure from Thibault and Naylor’s work, we do not worry about unioning or gluing together boundary fragments. Instead, we allow for boundary polygons within a sub-hyperplane to overlap. If a non-redundant boundary is needed (say for output) then we reconstruct a boundary, directly from the tree, for output. This change gives us one less thing to worry about. From the ’87 paper, we take the BSP tree reduction techniques. We found tree reduction to be very important for saving both time and space.

3.4. Interoperability: Input and Output



The system we have described is designed to work within a nice sanitized world of objects represented as plane-based BSP trees. Unfortunately, no other operations or systems are designed to work within this world. In order to accommodate these circumstances, we provide the following translations: between point and plane-based polygon soup, and then between plane-based polygon soup and BSP trees. In this way, input and output to and from the plane-based BSP tree representation may be decomposed into 4 simpler transformations.

point soup → plane soup. Each point-based polygon is approximated by a plane-based polygon. First, a plane is fit to the vertices of the point-based polygon, to serve as the plane of support s of the new plane-based polygon. Using the plane to polygon construction (§3.2), an initial polygon is created. Then for each edge of the point-based polygon, a new bounding plane is fabricated, passing through that edge and orthogonal to the plane of support s. The initial polygon is then successively clipped by these bounding planes, using the polygon splitting routine (§3.2).

plane soup → BSP tree. Because we are not guaranteed that the resulting plane-based polygon soup is watertight, we assume the worst and implement a mesh repair algorithm. Murali and Funkhouser have devised a clever mesh repair scheme, based on BSP trees [MF97]. The mesh repair scheme builds a consistent BSP tree as an intermediary step, and so is perfectly suited to this role. Alternatively, if we know that the plane-based polygon soup is water-tight (for instance, if it was a boundary extracted from a BSP tree), then we can use a standard BSP tree building algorithm, such as the one described by Thibault and Naylor [TN87].

BSP tree → plane soup. Since the boundary is already stored in the BSP tree, all we need to do is traverse the tree and collect it. As a minor note, we may want to recompute the boundary of the tree once before doing so for reasons previously mentioned(§3.3).

plane soup → point soup. Since the vertices of a plane-based polygon are implicitly defined as the intersection of 3 given planes (§3.2), we may convert to point-based output by computing/approximating the coordinates of these vertices.

connectivity. Although we never use or record connectivity information (as one would when working with B-reps) many other programs expect to find connectivity information encoded in their input polygon soup. All of this information may be recovered by knowing precisely when two vertices of two different polygons are actually the same point in space. Since any plane based polygon soup output from a BSP tree is guaranteed to be valid, we can recover this vertex incidence information from the plane soup using the orientation predicate. Given two valid points (a,b,c) and (x,y,z) (defined as triples of planes), (a,b,c) is incident to (x,y,z) if and only if (a,b,c) lies on all three planes x, y, and z.

accuracy. For our purposes (sculpting) accuracy was only important (a) up to visual inspection and (b) in order to achieve robustness. Towards this end we make a distinction between the accuracy of conversion (the consistency between the input and output of a given conversion step) and validity of output (the self-consistency of a representation). Note that the proposed I/O path always ensures validity. Therefore the program will not crash or fail. However, the I/O steps do not always guarantee 100% accuracy. In particular, conversion to and from point and plane based polygon soup is computed inexactly, making it subject to roundoff error. Numeric accuracy is only lost in these conversions. (This loss may be mitigated to machine epsilon in both input and output sides using Priest's techniques [Pri91], on which Shewchuk's predicates are based.) Additionally, the perturbation caused by roundoff during input will break every non-trihedral vertex (with degree $k > 3$) into $k-2$ nearly coincident, but distinct vertices, potentially introducing micropolygons and/or splinter-like polygons. Because we are only concerned with accuracy up to visual inspection, we can run a mesh simplification program on our output to collapse split vertices and eliminate degenerate polygons. Since this mesh simplification only relies on valid connectivity data to operate robustly, no new robustness issues are introduced.

4. Experiments

4.1. Setup

We ran four tests on our system (BSP), CGAL's [CGA08] Nef polyhedra, and Maya's [Aut09] Booleans. Under our taxonomy (§2) BSP uses an exact (fixed precision) BSP tree approach, CGAL an exact (arbitrary-precision) B-rep (technically Nef polyhedron) approach, and Maya a fragile B-rep approach. We ran all tests on a 1.83 Intel Core Duo (32bit) Macbook Pro. We compiled CGAL with static libraries, linked using gcc and options -O2 -DNDEBUG. CGAL kernels were *Exact_predicates_exact_constructions*. We used A trial version of Maya 2009. No paging, etc. occurred.

We report times for performing Boolean operations, whose definition varies from system to system. For BSP, this consists of computing the resulting tree with its boundary embedded(§3.3). For CGAL, this consists of computing the resulting Nef polyhedron. However, some extra computation (beyond lookup) is then necessary to compute the boundary. We did not measure this time since we felt it would unduly punish the design decision to use Nef Polyhedra. For Maya, MEL scripting was used to measure the time to execute a Boolean command. Input conversion times (only applicable to octoball) are provided separately from Boolean times.

4.2. Tests and Results

This battery of tests was chosen to give a good coverage of potential Boolean use in a sculpting system from single operations between two sizable meshes (Octoball) to iterated applications of small tool shapes (Random Boxes, Sculpt Bunny). We also included the heatsink test to check for worst case performance when the output is $O(n^2)$ in the size of the input.

Octoball Test The Octoball test computes a single set operation: a lumpy ball volume minus an octopus volume (Figure 7). The resulting ball is left with many tunnels and a few internal bubbles (octopus eyes). In addition, this test was useful as an "organic" rather than synthetic test, since the models were found rather than fashioned for this purpose. We scaled this test by applying mesh simplification to the octopus and ball models, yielding 10 copies with $0.1, 0.2, \ldots, 1.0$ times as many faces as the original meshes, which had 11444 faces (ball) and 33872 faces (octopus) respectively.

No systems failed on any instance of this test. BSP ran roughly $1.5\times$ slower than Maya and about 3-$6\times$ faster than CGAL. The input routines for BSP resulted in trees containing $4\times$ as many nodes as faces in the input model, due to microscopic and near degenerate polygons formed by point to plane conversion and the subsequent mesh repair algorithm. However, BSP still maintained a memory footprint smaller than CGAL: BSP took 30MB for the $0.1\times$ face meshes compared to CGAL's 50MB footprint. For the $1.0\times$ face meshes BSP took 250MB vs. CGAL's 270MB. (Memory usage was measured using process monitoring.) Furthermore, BSP took 20 seconds and 6 minutes respectively for the $0.1\times$ and $1.0\times$ conversions as compared to CGAL's 6 minutes and 2 hours.

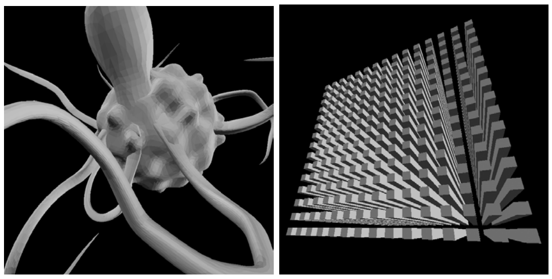


Figure 7: *(left) Octoball: the octopus is subtracted from the bumpy ball in the test. They are displayed unioned for visual reference; (right) Heatsink: the test result is displayed for $n = 15$; 225 rods*

Heatsink Test The heatsink test consists of computing a single intersection operation between n slabs arrayed along the x axis (vertical slabs) and n slabs arrayed along the y axis (horizontal slabs), resulting in n^2 rods (Figure 7). This test induces the worst case $O(n^2)$ behavior.

Maya produced incorrect results on this test. BSP was much faster than either Maya or CGAL (asymptotically so and by an average constant factor of $50\times$ over CGAL). We do not know why Maya's performance on heatsink appears to be exponential (see table), and we are loath to conjecture without further information.

Random Box Test The Random Boxes test consists of n iterative, varying unions and subtractions of randomly chosen rectangular prisms. The scene begins with the 0th box and each ith box thereafter is unioned into the scene unless i is divisible by 5 or 7, in which case it is subtracted. Box width is bounded between 1.0 and 3.0 in each dimension, positions are chosen randomly throughout a $10.0 \times 10.0 \times 10.0$ cubic volume, after which a random rotation is applied. This test provides a synthetic stress test for iterated Boolean operations, as one would use in sculpting. The same set of random boxes was used across all three tested systems.

Maya failed this test gracefully after about 90 operations. By periodically rebuilding the tree (from plane-based polygon soup) every 20 operations, we were able to get a 2-$3\times$ speedup on BSP. Furthermore, this dramatically reduced tree size, maintaing it beneath 20,000 nodes, rather than it proceeding upwards of 100,000. As a result, BSP was much faster than CGAL (22-$28\times$).

Sculpt Bunny Test The Sculpt Bunny test consists of automatically sculpting the Stanford bunny out of a block of "clay" by repeated application of boolean difference operations to subtract out smaller chunks. The chunks to be removed (dodecahedra) are placed by sampling the bunny volume with a regular grid and choosing a subset of the samples outside the bunny. Radii (dodecahedra scaling) are associated with each of these sample points so that the subtracted volume covers all of the samples outside of the bunny. This procedure was used to generate our Stanford Bunny model (Figure 1), using an originally 80^3 sample grid. For this test we used a 10^3 sample grid, ultimately resulting in 250 dodecahedra to subtract. This same set of dodecahedra was used across all three systems.

Maya successfully completed these 250 operations. Therefore we tried a larger 20^3 resolution yielding about 850 operations. On this test set, Maya gracefully failed after 25 operations. Again, we used tree rebuilding to accelerate BSP, yielding a $2\times$ speedup. With tree rebuilding, BSP was only $2\times$ slower than Maya. BSP was again much faster than CGAL (16-$17\times$).

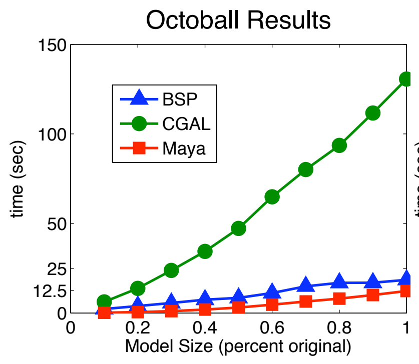
4.3. Analysis

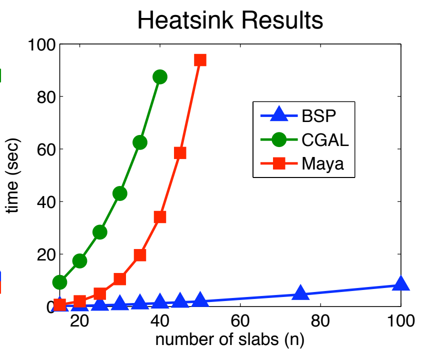
As a general trend, BSP was much faster (16-$28\times$ iterated tests, 3-$6\times$ octoball) than CGAL, our exact modeler, yet only 1.5-$2\times$ slower than Maya, our fragile modeler. Given that our code has not been optimized, we believe this provides strong evidence that we can build exact Booleans that run as fast as fragile Booleans. We have also definitively shown the excessive cost of arbitrary-precision exactness techniques for Booleans, even in the case of a single Boolean operation (octoball, heatsink).

5. Conclusion

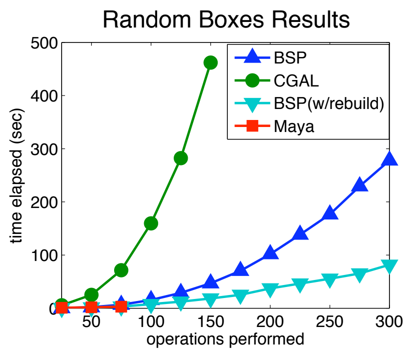
5.1. Discussion

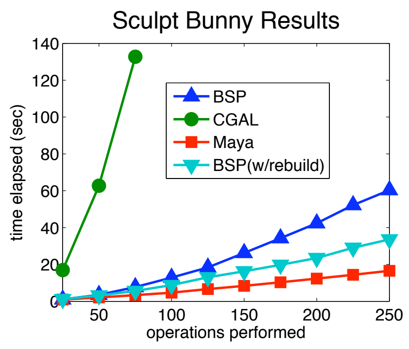
We presented our system design(§3) as if we selected the constituent parts independently: one algorithmic approach,

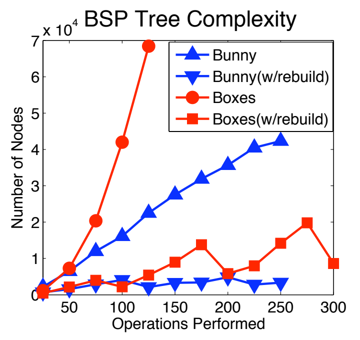




Heatsink Results			
	Time (seconds)		
n	BSP	CGAL	Maya
15	0.2	9.3	0.7
20	0.3	17.4	2.0
25	0.5	28.4	4.9
30	0.7	43.1	10.5
35	1.0	62.5	19.6
40	1.3	87.4	34.1
45	1.7	n/a	58.5
50	2.0	n/a	93.9
75	4.6	n/a	n/a
100	8.1	n/a	n/a







one substrate and one robustness approach. However, these choices are highly dependent on one another. In order to get robustness on the cheap, we use fixed-precision robustness, not arbitrary-precision. To do so we must avoid constructions, which requires using planes not points in our geometric substrate. Thus the interface provided to the algorithm must be plane-based, making BSP tree algorithms ideal since BSP trees are completely based on planes that partition space. In return the BSP tree algorithm requires an unusually concise substrate to handle all degeneracies. We collect these disparate reasons under the unifying maxim, "use planes, not points," a fundamental insight into the nature of computing Booleans.

Furthermore, swapping out a single component is not generally possible. Without exact arithmetic, BSP trees become intolerably fragile. Even plane-based B-rep algorithms yield complicated substrates. Demanding points forces the use of arbitrary-precision arithmetic.

5.2. Applications and Limitations

Although our system was designed for sculpting, it seems reasonable to use in more traditional 3d modelers to replace fragile Booleans. However, such systems are likely to use B-reps, not BSP trees. The described input and output paths(§3.4) provide a rudimentary sketch of the representation conversions needed, but there are two major limitations: the conversion is slow (a few seconds to a few minutes), and the result may be verbose and ill-conditioned (microscopic and near degenerate polygons). Given these limitations, our system could be most profitably used in a distinct "BSP tree" mode, where users explicitly convert models into BSP trees, perform set operations, and then convert back, allowing ample time for conversions.

Other limitations include restriction to linear geometry and not handling rotations. Rotation arithmetic induces roundoffs, inadvertently altering the arrangement of geometry. As a quick solution, it's possible to convert a BSP tree to plane-based polygons, rotate, and then convert back into a BSP tree (using mesh repair).

5.3. Future Work

As demonstrated (Figure 1) our system can compute Boolean operations at approximately 1Hz for reasonably complicated objects. One avenue for future work is trying to improve performance until robust Booleans are truly interactive for complex objects. We see further performance improvements coming in two ways: managing model complexity and special casing operations. Iterated operations tend to increase model complexity, slowing down both future modeling operations and rendering. This complexity manifests in tree size (number of nodes) and number of polygons. What sort of (possibly lossy) tree simplification can we perform while still supporting Boolean operations? How can we make operations faster? Can we exploit simplifying assumptions, such as: The tool object is small relative to the sculpted object, it is convex, or repeated applications are likely to happen close together?

Another avenue for future work is improving conversion between BSP trees and polygonal meshes. Again, making the problem more specific will help. Given a consistent point-based mesh, can we skip mesh repair on input and get a consistent BSP tree? Can we use partial conversions to safely and quickly support other operations like rotations or even free-form deformations?

References

[Aut09] AUTODESK: Maya, 2009.

[BMP93] BENOUAMER M., MICHELUCCI D., PEROCHE B.: Error-free boundary evaluation using lazy rational arithmetic: a detailed implementation. In *SMA '93: Proceedings on the second ACM symposium on Solid modeling and applications* (New York, NY, USA, 1993), ACM, pp. 115–126.

[BR96] BANERJEE R. P., ROSSIGNAC J. R.: Topologically exact evaluation of polyhedra defined in csg with loose primitives. *Computer Graphics Forum 15*, 4 (1996), 205–217.

[Bru91] BRUDERLIN B.: Robust regularized set operations. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences* (1991), vol. 1, IEEE, pp. 691–700.

[CGA08] CGAL, Computational Geometry Algorithms Library, 2008. http://www.cgal.org.

[DP03] DEVILLERS O., PION S.: Efficient exact geometric predicates for delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.* (2003), pp. 37–44.

[FCG00] FERLEY E., CANI M. P., GASCUEL J.-D.: Practical volumetric sculpting. *The Visual Computer 16*, 8 (Dec 2000), 469–480.

[For97] FORTUNE S.: Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design 29*, 2 (1997), 123–133.

[FVW96] FORTUNE S., VAN WYK C. J.: Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph. 15*, 3 (1996), 223–248.

[GH91] GALYEAN T., HUGHES J.: Sculpting: an interactive volumetric modeling technique. *SIGGRAPH Comput. Graph.* (Jul 1991).

[GHH*03] GRANADOS M., HACHENBERGER P., HERT S., KETTNER L., MEHLHORN K., SEEL M.: Boolean operations on 3d selective nef complexes: Data structure, algorithms, and implementation. In *ESA* (2003), pp. 654–666.

[HHK89] HOFFMANN C. M., HOPCROFT J. E., KARASICK M. E.: Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl. 9*, 6 (1989), 50–59.

[HK05] HACHENBERGER P., KETTNER L.: Boolean operations on 3d selective nef complexes: optimized implementation and experiments. In *SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling* (New York, NY, USA, 2005), ACM, pp. 163–174.

[Hof89] HOFFMANN C. M.: *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[LDS09] LYSENKO M., D'SOUZA R., SHENE C. K.: Improved binary space partition merging. *Computer Aided Design* (2009). (to appear).

[LTH86] LAIDLAW D. H., TRUMBORE W. B., HUGHES J. F.: Constructive solid geometry for polyhedral objects. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 161–170.

[MF97] MURALI T. M., FUNKHOUSER T. A.: Consistent solid and boundary representations from arbitrary polygonal data. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics* (New York, NY, USA, 1997), ACM, pp. 155–ff.

[NAT90] NAYLOR B., AMANATIDES J., THIBAULT W.: Merging bsp trees yields polyhedral set operations. In *SIGGRAPH Comput. Graph.* (New York, NY, USA, 1990), ACM, pp. 115–124.

[Nay90] NAYLOR B.: Sculpt: an interactive solid modeling tool. In *Proceedings on Graphics interface '90* (Toronto, Ont., Canada, Canada, 1990), Canadian Information Processing Society, pp. 138–148.

[Nef78] NEF W.: *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, 1978.

[PF01] PERRY R., FRISKEN S.: Kizamu: a system for sculpting digital characters. *SIGGRAPH Comput. Graph.* (Aug 2001).

[Pri91] PRIEST D. M.: Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic* (1991), IEEE, pp. 211–215.

[Req77] REQUICHA A. A. G.: *Mathematical Models of Rigid Solid Objects*. Tech. Rep. TM-28, Production Automation Project, University of Rochester, Rochester, New York 14627, November 1977.

[RV85] REQUICHA A., VOELCKER H.: Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE 73*, 1 (1985), 30–44.

[Seg90] SEGAL M.: Using tolerances to guarantee valid polyhedral modeling results. In *SIGGRAPH Comput. Graph.* (New York, NY, USA, 1990), ACM, pp. 105–114.

[Sei94] SEIDEL R.: The nature and meaning of perturbations in geometric computing. In *STACS '94: Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science* (London, UK, 1994), Springer-Verlag, pp. 3–17.

[She97] SHEWCHUK J. R.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry 18*, 3 (Oct. 1997), 305–363.

[She06] SHEWCHUK J. R.: Lecture notes on geometric robustness. Available online, Oct 2006.

[SI89] SUGIHARA K., IRI M.: A solid modelling system free from topological inconsistency. *J. Inf. Process. 12*, 4 (1989), 380–393.

[Sto87] STOLFI J.: Oriented projective geometry. In *SCG '87: Proceedings of the third annual symposium on Computational geometry* (New York, NY, USA, 1987), ACM, pp. 76–85.

[TN87] THIBAULT W. C., NAYLOR B. F.: Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 153–162.

[Yap04] YAP C. K.: Robust geometric computation. In *Handbook of discrete and computational geometry*, Goodman J. E., O'Rourke J., (Eds.). CRC Press, Inc., Boca Raton, FL, USA, 2004, ch. 41, pp. 927–952.

[YN97] YAMAGUCHI F., NIIZEKI M.: Some basic geometric test conditions in terms of plücker coordinates and plücker coefficients. *The Visual Computer 13*, 1 (Feb 1997), 29–41.