PHYSOR 2012 – Advances in Reactor Physics – Linking Research, Industry, and Education
Knoxville, Tennessee, USA, April 15-20, 2012, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2012)

ROBUST VOLUME CALCULATIONS FOR CONSTRUCTIVE SOLID GEOMETRY (CSG) COMPONENTS IN MONTE CARLO TRANSPORT CALCULATIONS

David L. Millman, David P. Griesheimer, Brian R. Nease, and Jack Snoeyink
Department of Computer Science
University of North Carolina at Chapel Hill
{dave, snoeyink}@cs.unc.edu

Bechtel Marine Propulsion Corporation
Bettis Atomic Power Laboratory
{david.griesheimer, brian.nease}.contractor@unnpp.gov

ABSTRACT

In this paper we consider a new generalized algorithm for the efficient calculation of component object volumes given their equivalent constructive solid geometry (CSG) definition. The new method relies on domain decomposition to recursively subdivide the original component into smaller pieces with volumes that can be computed analytically or stochastically, if needed. Unlike simpler brute-force approaches, the proposed decomposition scheme is guaranteed to be robust and accurate to within a user-defined tolerance. The new algorithm is also fully general and can handle any valid CSG component definition, without the need for additional input from the user. The new technique has been specifically optimized to calculate volumes of component definitions commonly found in models used for Monte Carlo particle transport simulations for criticality safety and reactor analysis applications. However, the algorithm can be easily extended to any application which uses CSG representations for component objects. The paper provides a complete description of the novel volume calculation algorithm, along with a discussion of the conjectured error bounds on volumes calculated within the method. In addition, numerical results comparing the new algorithm with a standard stochastic volume calculation algorithm are presented for a series of problems spanning a range of representative component sizes and complexities.

Key Words: Monte Carlo, Constructive Solid Geometry, Octree, Robust Computation, Volume Calculation.

1. INTRODUCTION

One of the major strengths of Monte Carlo (MC) particle transport methods is their ability to model complex geometries with a high degree of fidelity. Monte Carlo methods that use constructive solid geometry (CSG) representations for modeling component objects offer nearly unlimited flexibility for model construction. In addition, CSG models allow most curved surfaces (typically up to second-order) to be represented exactly, without the need for the spatial discretization required by mesh-based transport solvers. This level of geometric detail is frequently desired when dealing with complex models for criticality safety and reactor analysis applications, and, as a result, the CSG representation has been used in several major MC transport solvers, such as MC21 [1] and MCNP [2]. Through taking Boolean operations (union, intersection, and set difference) of surface half-spaces, the CSG representation allows users to construct many component objects more accurately than with a polygonal mesh.

Ray tracing through CSG components involves computing the distance to each bounding surface of the component object along a given ray and then applying simple Boolean logic operations to determine which surface intersection will occur first along the ray. This does not require explicitly representing the boundary of the component object, but, as a result, ray tracing for CSG representations is more computationally expensive than for mesh-based geometries.

While CSG representations allow MC codes to support nearly unlimited flexibility in model creation, it leads to other issues such as detecting invalid object definitions (e.g., overlapping components), lost particles during ray tracing (e.g., near surface intersections), and computing the volume of components. While many work-around solutions have been found for these issues, they still continue to plague MC calculations with CSG component representations.

Initially, MC calculations were primarily used to compute the global eigenvalues. As a result, emphasis was placed on resolving particle tracking (ray tracing) errors, as opposed to computing component volumes. While component volumes are often used as a model quality assurance check, their calculation does not influence the radiation transport calculation itself. However, recent efforts to include in-line feedback effects (e.g. depletion, thermal feedback, xenon feedback, etc.) in MC reactor calculations have provided a need for MC solvers to calculate component volumes with a high degree of accuracy. While some popular MC codes, such as MCNP [2] calculate the volumes of simple component objects, no code available today is able to guarantee an accurate volume calculation for all possible CSG component definitions involving second-order surfaces. In cases where codes cannot calculate component volumes analytically, users are typically required to input a volume for the component, which must be obtained from some external calculation or *a priori* information.

For complicated CSG component definitions, the options for computing volumes robustly are limited and fairly simplistic. Analytical volume calculation methods may be used for objects with a simple two-dimensional projection (such as extruded components, or those with a rotational symmetry), but are impractical for general three-dimensional components. Stochastic (Monte Carlo) methods rely on sampling points within a fixed volume that is known to bound the object of interest and then applying a binomial estimator to estimate the component volume (relative to the known volume of the bounding box). Alternatively, discretization techniques may be applied. However, such approaches require the construction of a mesh representation for the original CSG object, which is, itself, a formidable task for complex component definitions. Unfortunately, both the stochastic and discretization approaches to volume computation are computationally intensive and are too slow for routine use, such as for an in-line volume calculation module in a MC transport solver.

In this paper we consider a new generalized algorithm for the efficient calculation of component volumes, given their equivalent CSG definition. The new method relies on domain decomposition to recursively subdivide the original component into smaller pieces with volumes that can be computed analytically or stochastically, if needed. Unlike simpler brute-force approaches, the proposed decomposition scheme is guaranteed to be robust and accurate to within a user-defined tolerance. The new algorithm is also fully general and can handle any valid CSG component definition, without the need for additional input from the user.

The volume calculation algorithm is actually a framework, based on the concept of decomposability and the use of integrator modules that are designed to handle volume calculations for simple base cases. In this paper, we present an outline for this framework, along with a specific example of a robust decomposition scheme that uses representative integrators designed to handle the size, and potential complexity, of components found in MC radiation transport models. Through a series of controlled numerical experiments we demonstrate that the new algorithm can produce accurate volume estimates for complex components to within specified accuracy and confidence bounds. Furthermore, the results from these experiments indicate that the new method can produce these volume estimates up to several hundred times faster than traditional brute-force volume calculation methods.

This paper attempts to provide a solid theoretical foundation for CSG component representations and the proposed volume calculation algorithm. Section 2 of the paper describes a standard CSG representation of MC components, using up to second-order (quadric) surfaces, and discusses the use of component hierarchy in models. This section also provides precise mathematical definitions for surfaces and components, which are used throughout the remainder of the paper. Section 3 describes a robust domain decomposition algorithm that can be used to subdivide a given component into a set of simpler objects. This section also defines a set of efficient predicate operations to determine how many intersections occur between surfaces of a component and an arbitrary, axis-aligned test box. These predicates are used to classify partition volume boxes created during the decomposition process. Section 4 describes a set of efficient volume integration algorithms, which have been optimized to certain base-case volumes. These base volume cases correspond to many of the surface-box intersection classifications described in Section 3. It should be noted that the divide and conquer strategy for volume computation employed in this paper is a well-known paradigm in computer science. The novelty of this new framework is that we aim for a minimal set of predicates (tests that control branching) on surfaces, which not only ensures robustness by limiting numerical computation to within surfaces, but also facilitates speed and accuracy improvements by specially coding common cases, as we demonstrate through experiments reported in Section 5.

In addition, this paper is intended to provide a brief introduction to some of the concepts and terminology used in the field of computational geometry. During the early stages of this research it was discovered that the computational geometry/computer graphics community has already looked at many problems of interest for MC radiation transport, especially with respect to efficient algorithms for ray tracing through CSG representations. As a result, algorithm descriptions and related proofs (for completeness, robustness, accuracy, etc.) will follow conventions and terminology adopted from computational geometry. It is our hope that this introduction will promote additional collaboration and flow of information between the two technical communities.

While the example components used for the numerical experiments in this paper are intended to be representative of complex components used in reactor and criticality safety analysis calculations, the algorithm itself can be easily extended to any application which uses CSG representations for component objects. In addition, the performance of the new method is approaching a point where in-line volume calculations for all components in MC models may become feasible. Furthermore, the data structures and general algorithms described in this paper may also have additional applications beyond volume calculations. For example, the new algorithm can be used to produce axis-aligned bounding boxes, which are guaranteed to include all points located inside of a given component definition. The use of bounding boxes to accelerate ray tracing to components is a standard technique within the computer graphics community, but has seen limited use in MC particle transport calculations due to the difficulty in computing bounding boxes that are guaranteed to be correct. Therefore, it is expected that the algorithm presented in this paper will also lead to future improvements in tracking efficiency.

2. MODEL REPRESENTATION

Many Monte Carlo codes that solve the neutron transport equation, such as MC21 [1] and MCNP [2], use a constructive solid geometry (CSG) or combinatorial geometry representation to define regions of space. A combinatorial geometry model combines basic spatial primitives, bounded by surfaces (planes, cylinders, spheres, ellipsoids, etc.), using Boolean operations (union, intersection, complement, difference). A model consists of one or more interior disjoint, closed regions, called components, which may be organized in a parent/child hierarchy to reduce memory overhead and to accelerate tracking. We give concrete details for the surfaces, Boolean formulae, and components, as well as the interface that these objects should provide.

2.1. Primitives: Signed Quadric Surfaces

A primitive of the model is a signed surface that represents the points in one of the regions that it bounds. To be more specific, we consider quadratic surfaces.

A signed quadric Q consists of a quadric polynomial $q(x, y, z)$ and a sign bit. The zero set of this polynomial is the points on the surface Q. The set $\{(x,y,z) \mid q(x,y,z) < 0\} \subseteq \mathbb{R}^3$ is the *negative halfspace* of Q, and $\{(x,y,z) \mid q(x,y,z) > 0\}$ is the *positive halfspace*. A point $p \in \mathbb{R}^3$ has a *negative sense* with respect to Q if p is in the negative halfspace of Q, and a *positive sense* if p is in the positive halfspace. The sign bit for Q indicates which halfspace the primitive represents.

Quadric Q is traditionally stored as ten single precision floating point coefficients, A through J of

$$q(x,y,z) = Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J. \tag{1}$$

Positive scalar multiples of Eq. (1) give the same surface, while negative multiples reverse the positive and negative senses. In addition, many "natural quadrics," have more compact representations, such as,

1. Planes by quadrics with $A = B = C = 0$,
2. Spheres by center (A, B, C) and radius D or as a quadric: $(x-A)^2 + (y-B)^2 + (z-C)^2 - D^2 = 0$,
3. Axis aligned ellipsoids by the form: $A(x+B)^2 + C(y+D)^2 + E(z+F)^2 + G = 0$,
4. Cylinders orthogonal to the *xy*-plane: $A(x+B)^2 + C(y+D)^2 + E = 0$.

As an abstract data type, a surface must provide a point test, which identifies whether a point *p* has the same sense as the surface sign, and, thus, is in the primitive. For our framework, the surface must also provide a box classification test, which reports whether all points in an axis-aligned box have the same sense, or opposite sense, or both, with respect to the surface. The surface can optionally provide an integrator that calculates the volume of the primitive intersected with a box.

2.2. Component Hierarchy: Boolean Formulae

In many solid modeling tools, primitives are combined by Boolean operations[1] (union, intersection, difference, and complement) to describe more complicated regions. A multi-component model is described as a tree hierarchy in which each node represents a component defined by Boolean operations, which is contained in its parent, disjoint from its siblings, and contains its children.

To be precise, there are three types of components that pertain to an object represented as a node *N* in the hierarchy tree. A simple example is illustrated in Figure 1.

- The *basic component B(N)* is a region defined as a DNF (Disjunctive Normal Form, i.e., a union of intersections) formula of primitives that are bounded by quadric surfaces. *B(N)* is a blue rectangle in the figure. The requirement that the region be defined in DNF is not limiting, since any region definition may be rewritten in DNF.

- The *restricted component R(N)*, drawn as a purple-striped rectangle, is the intersection of *B(N)* with the restricted component of its parent. Thus, restricted components nest, with the one at the root, N_0, always containing the entire model. The interiors of restricted components of siblings of *N* are assumed to be disjoint from that of *N*; that is, sibling components on the same level of hierarchy are not allowed to overlap. If we explicitly include the intersection of ancestors, a restricted component has a 3-level formula (i.e., an intersection of unions of intersections).

- The *hierarchical component C(N)*, striped purple and blue, is the restricted component *R(N)* minus the restricted components of the children of *N*. We seek to compute the volumes of all hierarchical components in the tree.

As an abstract data type, the Boolean formulae and hierarchy tree must provide point location, which locates the restricted component containing a query point *q*, and *formula restriction to a box* which takes the results of the box classification test for all primitives and returns simplified formulae and tree by replacing primitives that are entirely outside or containing the box with false or true, respectively, and applying logical rules to obtain a equivalent formula without these logical constants. Note that these operations are entirely logic and data structure manipulation. As all the numeric evaluations are performed in the surfaces, these operations cannot introduce any numerical errors.

[1] Often by regularized operations [3], which follow the operation by taking the boundary of the interior. Regularization does not change volume, but does remove lower dimensional features from an object.

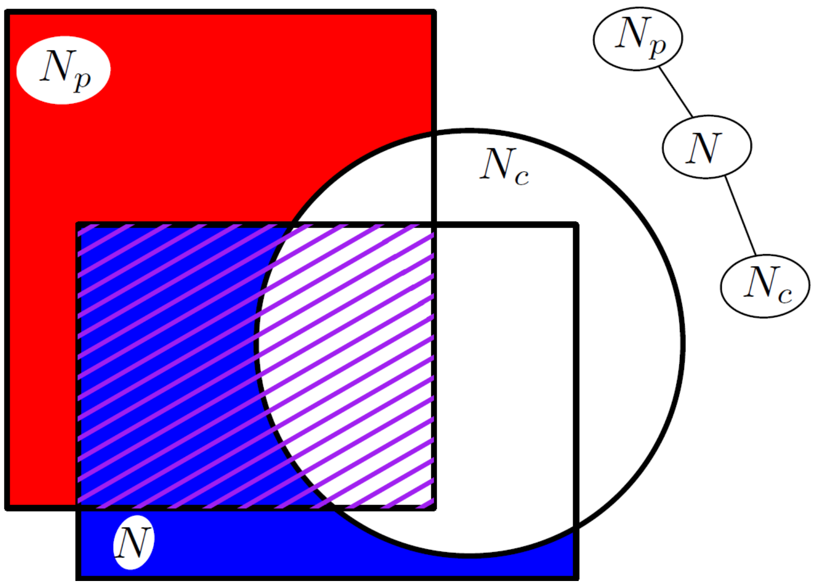

Figure 1. Hierarchy N_p – N – N_c with component $C(N)$ striped purple and blue.

2.3. Problem Statement

The tree also drives the process of volume computation for the components, and we ask it to provide the *volume of each restricted component*. The volumes of the hierarchical components are then computed by having each parent subtract the volumes of its children.

To obtain these volumes we extend the notion of an integrator from a surface to the entire hierarchy of components: an *integrator* is supposed to provide the volumes of a given box intersected with each restricted component in a hierarchy tree. Since it will do so by creating a spatial subdivision, referred to as an octree, which is a third hierarchical structure, we find the following sections more clear if we consider the problem of a single restricted component, so that we have only a 3-level Boolean formula and an octree to distinguish between. Thus, in the following sections we focus on Problem 1. In fact, however, we evaluate all restricted components in the hierarchy tree together, which is important to prevent the subtractions from compounding errors.

Problem 1 *Given an axis-aligned bounding box BB and a restricted component R, defined as a 3-level formula on signed surfaces* $\{S_1,..., S_n\}$*, compute the volume of their intersection,* $BB \cap R$*, to specified precision.*

3. DIVIDE AND CONQUER USING PREDICATES

Here is a brief top-down description of the *divide and conquer integrator* for computing the volume of a restricted component (actually, the entire component hierarchy, as suggested in Section 2) using predicates mentioned in the previous section, and described in detail below.

The input is the Boolean formula on a set of surfaces defining primitives, and an initial axis-aligned bounding box. The procedure depends on a set of integrators for base cases, such as boxes that intersect a couple of surfaces; these are described in Section 4. The formula is first simplified by restriction to the box, which involves applying box classification for all surfaces; these predicates are detailed in the remainder of this section. If it is then simple enough, or if the box is small enough to apply a base case integrator, then this completes the computation. Otherwise the box is subdivided and copies of the formula are given to each box; we use a subdivision into eight boxes, forming an octree [4] for our implementation and experiments. Evaluation on smaller boxes can be performed in parallel, as the only dependency is the summation of return values.

3.1. Surface/Axis-Aligned Box Classification

Before going into the details of the box classification test we recall the conic classification of Levin [5], and derive a method for sampling a point inside an ellipse or ellipsoid.

3.1.1. Primitives of the box classification test

Recall that the intersection of a quadric and a plane is represented by the zero set of a conic, a polynomial of degree 2 over two variables:

$$c(x,y) = Ax^2 + By^2 + Dxy + Gx + Hy + J = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} A & D/2 & G/2 \\ D/2 & B & H/2 \\ G/2 & H/2 & J \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = 0 \qquad (2)$$

Conics represent ellipses, hyperbolas, parabolas, lines, etc. Levin [5] describes how to classify a conic by properties of its *discriminant*, the 3 × 3 symmetric matrix shown in Eq. (2). This classification is summarized in Table I. Levin describes how to classify quadrics as well; we use his test when initializing a quadric to identify ellipsoids and sample a point from inside each one.

Table I. Classification table for conics with real coefficients.

d	m	s	Condition	Classification
1	1	1		Coincident lines
2	0	0		Single line
2	1	1	$\delta < 0$	Two parallel lines
2	2	0	$\|Q_u\| < 0$	Two intersecting lines
2	2	2	$\|Q_u\| > 0$	Point
3	1	1		Parabola
3	2	0	$\|Q_u\| < 0$	Hyperbola
3	2	2	$\|Q_u\| > 0$ and $\|Q\|t<0$	Ellipse
	else			Invalid or imaginary conic

For a conic given in matrix form (Eq. 2), let Q be the discriminant and d be the rank of the matrix. Let Q_u be the upper left 2×2 submatrix, with rank m, signature s (defined as the number of positive eigenvalues minus the number of negative eigenvalues), and trace t (defined by $t = A + B$). Finally, let $\delta=4(AB+AJ+BJ) - D^2 - H^2 - G^2$.

Silhouette curves, which help define bounding boxes and integration domains and help in sampling points inside bounded surfaces and curves, are also conics. We can write a quadric as a function of z, as $q(x, y, z) = az^2 + b(x,y)z + c(x,y)$, where $b(x,y)$ is a linear function and $c(x,y)$ is quadratic. The *silhouette curve* is the locus of double roots of z, which occur at x, y positions where $b(x,y)^2 = 4ac(x,y)$, and have $z = -b(x,y)/(2a)$. For quadrics the silhouette curves in coordinate directions lie in easily derived planes since, for example, the silhouette curve satisfies $\frac{d}{dz} q(x,y,z) = (2Cz + Ex + Fy + I) = 0$, and all such points satisfy $z = -(Ex + Fy + I)/(2C)$. This plane separates the surface into two z-monotone pieces. The projection of the silhouette curve of a quadric onto the xy-plane is $r(x,y) = q(x,y,-b(x,y)/(2a))$. For an ellipse $c(x,y)$, the silhouette curve is two points, s_1 and s_2, that lie on the line $\frac{d}{dy} q(x,y) = 2By + Dx + H = 0$ that separates the ellipse into two x-monotone curves. Their average, $(s_1 + s_2)/2$, is a point inside. For an ellipsoid, we can sample a point in the silhouette curve and raise it to the plane through the silhouette curve.

3.1.1. The box classification test

Given an axis-aligned box, B, and a signed surface, Q, that defines a primitive, we want to label box B as *Inside* if all its points have the same sense as the surface, *Outside* if all its points have the opposite sense as the surface, or *Intersecting* if it contains points of both senses. Below we summarize the box classification test. We implemented specific cases for a plane, axis-aligned cylinder, and the general form of a quadric.

- For planes, the box classification can be done easily by having Q perform point tests on all corners of B. However, for other quadrics we need to test intersections with box edges and faces, and even containment for bounded primitives. This is especially important because we will use this as the basic test to simplify a formula while traversing the octree. If we were to misclassify a surface (say, it only passed though the faces of a box) it would be simplified out of the formula too early, perhaps losing a surface that should be integrated by an ancestor.

- Axis-aligned cylinders are common enough that we test them by projecting along their axes.

- For general quadrics, edge tests assign values from box corners to pairs of variables in a quadric, and evaluate the discriminant of the third variable. This algebra computes the vertex tests along the way. If these tests identify the box as Intersecting, we return that label. Otherwise, we must continue to test faces and containment since, for example, a pipe could intersect a box boundary only at faces.

  To test if a face f intersects surface Q, we classify the intersection of Q with the axis-parallel plane P that contains f. If this curve is not an ellipse, it is unbounded, and since we already know that vertices and edges around f do not intersect Q, we know the face also does not intersect Q. If it is an ellipse, then we test a point from the interior of the ellipse to detect intersection with face f. Again, if we ever detect Intersecting, we return that label immediately.

Finally, if *Q* is an ellipsoid, we test a sample point inside to ensure it has the same sense as all corners, and report the final classification label for *B*.

3.2. Component/Box Classification and Restriction

Extending the classification of a box *B* from surfaces to components is a purely logical operation, and the restriction of the 3-level formula to *B* is just data structuring.

The input is the box *B* and the restricted component's 3-level formula *R*, which involves surfaces already classified with respect to *B*. We can apply the formula to conservatively combine the labels of *B* by the following rules, where we have abbreviated *X*: Intersecting, *I*: Inside, and *O*: Outside, and α: any.

complement $\overline{X} = X$, $\overline{I} = O$, and $\overline{I} = O$.
union $X \bigcup X = X$, $O \bigcup \alpha = \alpha$, and $I \bigcup \alpha = I$.
intersection $X \bigcap X = X$, $O \bigcap \alpha = O$, and $I \bigcap \alpha = \alpha$.

This is a conservative extension because the union of two objects that intersect box *B* may actually completely cover *B*, or their intersection within *B* may be empty. By classifying all such cases as Intersecting, however, the octree will simply refine the box and retest. Thus, the result can be guaranteed correct to the level of refinement.

To avoid classifying every child box for every surface, we need to simplify formula *R* by restriction to box *B*. We describe how this can be done recursively, while exploring the octree in depth-first order. At any step we have a box *B* and the 3-level formula for $B \cap R$ that uses only surfaces that were deemed Intersecting with respect to the parent of *B*.

Initially, we begin with a bounding box and the entire formula, represented as a tree of height three. (Recall that *R* is $\cap$ of $\cup$ of $\cap$ of surface primitives; each operator is represented by a list of pointers to the formulae or variables that it operates on.) We evaluate *B* with respect to each surface remaining in the formula. If *B* is Inside or Outside any surface, we apply the above union/intersection operations (with commutativity and associativity) to simplify the formula. If the formula simplifies to Inside or Outside, or if we decide for some other reason to stop traversing at box *B*, then we are done. Otherwise, we refine box *B*, pass the simplified formula to each child, and recursively evaluate each child.

In this recursive algorithm it is imperative that any surface that intersects a box *B* must be detected as intersecting the parent of *B*. For example, if faces were not tested in the box classification test and a cylinder was missed because it pierced only a face of *B*, then it would be simplified away, and not be evaluated in any children of *B*.

4. VOLUME ESTIMATORS

An *integrator* determines if the geometry inside a box is sufficiently simple that it can evaluate the volumes in a component hierarchy; integrators serve as the base cases for the recursive divide-and-conquer algorithm.

Each integrator receives as input: a box, an error interval and confidence value expressed either in absolute terms or relative to the volume of the box, and a component hierarchy. It can look at the size of the box, the number and kind of surfaces in the hierarchy, and its task is to evaluate the volumes of the relative components in the box to within the error interval with given confidence. We first describe several possible integrators, and then return to the analysis of error intervals. As a part of the description, each integrator is given an abbreviation, which is used to refer to the integrator throughout the rest of the paper.

The Box Integrator (Box) - The simplest integrator assigns a user-specified fraction of box volume to each component that can intersect a box in the spatial hierarchy. This is exact when no surface passes through a box, since then the box is entirely inside one component. Otherwise the confidence is directly proportional to the volume of the box, so this can always be a fallback integrator when boxes become small.

The General Integrator: Monte Carlo (MC) - A Monte Carlo integrator simply performs point-in-component tests. We obtain confidence bounds using the Wilson test [6], which, at the 95% confidence level suggests that the volume *v* of a component that receives *x* out of *n* samples is in the confidence interval

$$CI_{95} = \frac{x+2}{n+4} \pm \frac{2\sqrt{x(1-x)/n+1}}{n+4} \tag{3}$$

We estimate the volume as $v = x/n$ so that the total volume within the box is conserved.

Single Plane Integrator (1pl) - When a single surface defines the portion of a component in an axis aligned box, by some case analysis (often a non-trivial amount) we can derive the domains of integrals for the component's volume, which we can then evaluate numerically or analytically. Single planes are the easiest and most frequent case, and their implementation shows benefits immediately.

Pairs of Planes Integrator (2pl) - Extending the single quadric integrator to a pair is also possible. Again, the biggest payoff for the effort comes from handling pairs of planes. By implementing a special integrator for this case, any box that contains two planes becomes a base case and needs not be further subdivided.

Capped Cylinder Integrator (Cyl) - As elliptic cylinders arc infinite in one direction, they are often truncated by some other quadric (usually a plane). We found that implementing a special integrator for an axis-aligned cylinder truncated by a plane orthogonal to its axis showed benefits. Non-truncated axis aligned cylinders are also handled by this integrator.

Bundle of Cylinders Integrator (Bun) - Because bundles of axis-aligned pins and cylindrical pipes and vessels are so common in models, we found it worthwhile to implement a special integrator for pairwise-disjoint collections of axis-aligned cylinders (capped and uncapped) with circular cross sections. Intersecting cylinders are reduced to this case by subdivision.

4.1. Error Bounds

One concern that arises with the spatial subdivision of the problem is how to combine error bounds from individual boxes. In the worst case, bounds on the errors from each box would have to sum: an error of $\pm\delta$ from K boxes would become an error of $\pm\delta K$. This worst case occurs if all errors are not independent and occur in the same direction, perhaps by a rounding problem in an integrator. Careful coding is needed to avoid accumulation of such error.

We get an average case if the errors are independent; the error bound from a combination would grow more like $\pm\delta\sqrt{K}$. The easiest illustration of this is to imagine a Monte Carlo volume estimate that chooses random points in each box and decides if they are inside or outside the volume, giving independent Bernoulli trials. It could instead choose points from the union of all boxes with the variance going from $np(1-p)$ for each of K boxes to $Knp(1-p)$ overall, so the standard deviation increases by $\sqrt{K}$.

The best case is also relevant — dependent errors can cancel if they are made in opposite directions, and we actually benefit from that by evaluating the entire component hierarchy at once. Consider Monte Carlo estimates again; if we determine which component contains a trial point, then the estimates to volume will at least add up to the volume of the box (up to machine precision) so that the volume lost by one component will be gained by another and the total volume will be conserved.

5. EXPERIMENTS

In this section we demonstrate how adding integrators in our framework can reduce the running time and/or increase the accuracy of volume computation on a set of models shown in Figure 2. Timings are from a 3.2GHz Intel Xeon processor with 12GB RAM running Ubuntu 10.04.

We ran several experiments on each of five models, four of which are shown in Figure 2 (the image of the fifth model is omitted as it is visually similar to the images in the bottom row). Each model consists of a single component enclosed within a cube of unit volume. In all models, the units of length and volume are arbitrary, but may be assumed to be in cm and cm^3, respectively, for reader convenience. The first model is a randomly rotated $0.5 \times 0.5 \times 0.5$ cube (referred to as the "Cube" model), with volume approximately 0.125 (randomly rotating the cube induces a small error, reducing the volume by approximately 1×10^{-7}). We use this example to assess the accuracy of combinations of plane and Monte Carlo (MC) integrators. The second model is a union of intersections of capped cylinders with drilled holes (referred to as the "DrillCyl" model). We use this example to assess the performance of combinations of integrators on models with curved primitives. The third model is a wedge with twelve L-shaped pipes of radius 0.035 removed, defined by 55 surfaces and 31 unions of intersections (referred to as the "cPiped12" model). We also consider larger models based on the cPiped12 model, but with different

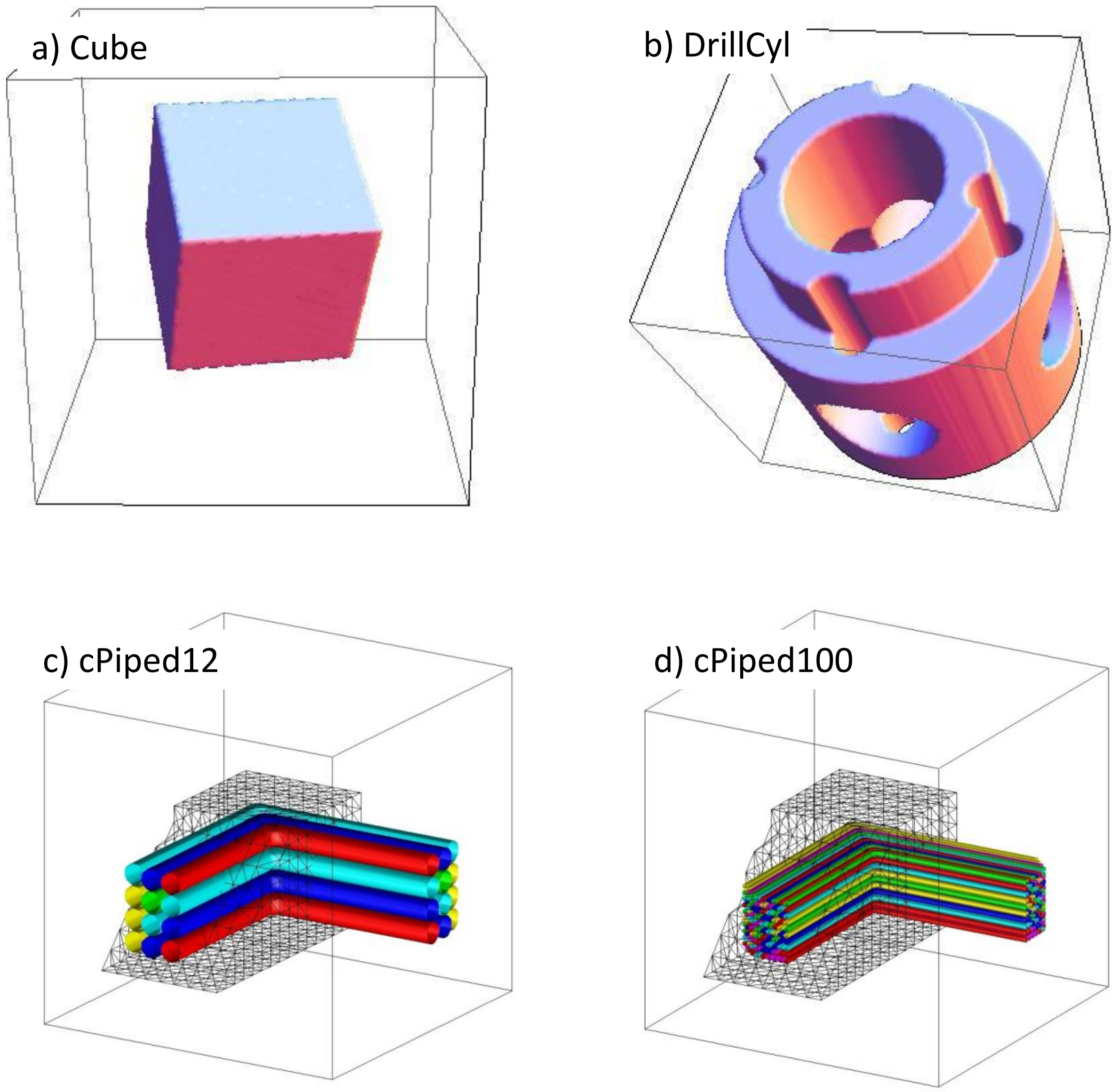

Figure 2. Models used for testing of volume calculation algorithm. a) Cube model is ≈ 0.125 volume cube with a random rotation; b) DrillCyl model is a union of two intersections of a dozen cylinders and planes; c) cPiped12 model is a wedge-shaped block minus a 3×4 bundle of L-shaped pipes; d) cPiped100 is similar to cPiped12 but with a 10×10 bundle of L-pipes; cPiped10000 (not shown) is similar to cPiped12 but with a 100×100 bundle of L-pipes.

numbers of pipes. These additional models include the “cPiped100” model, which contains 100 L-shaped pipes of radius 0.01, defined by over 475 surfaces and 200 unions of intersections, and the “cPiped10000” model, with 10,000 L-shaped pipes of radius 0.001, defined by over 40,800 surfaces and 20,400 unions of intersections.

For each model we evaluated volume with tolerances of $\pm 2.2\times10^{-3}$ and $\pm 1.1\times10^{-4}$. In our current implementation, tolerance is used primarily to determine the number of MC samples needed over the unit volume, which, in turn, determines the maximum depth of our octree, since the number of samples in a box is proportional to its volume, and we require at least eight points in a box to subdivide it. Thus, with tolerance 2.2×10^{-3} we stop at depth 6, and with 1.1×10^{-4} at depth 11. Descending further can change which integrators run. For example, in cPiped100 with tolerance $\pm 2.2\times10^{-3}$ Cyl integrates 7.6% of the boxes in volume calculation. However, when we explore the tree further with tolerance $\pm 1.1\times10^{-4}$, we find that Cyl integrates about 35.7% of the boxes. Because Cyl only integrates capped and uncapped single cylinders we need to travel further down the tree before we isolate a cylinder.

Tables II and III present results from the numerical experiments described above. For each model considered, each line of Tables II and III reports statistics of a run using all the integrators above that line, which explains the triangular pattern of numbers (e.g., the last line for each model uses all integrators). Table II reports the number of boxes on which integrators are run, and reports, for each type of integrator, the percentage of boxes that it evaluated. The number of boxes has a big effect on running time, but it does not tell the whole story. Table III shows the drastic decrease in the number of samples required by the MC integrator as some of its boxes are given to other integrators (the number of samples is directly proportional to the volume of the boxes in which MC integration is performed). Adding other integrators decreases the number of boxes for the MC integrator by a factor of 2, but the volume of boxes integrated by MC decrease by factors of more than 10. An integrator that is added but not used in many boxes can still reduce the number of boxes dramatically by serving as an early base case.

We observe that in all cases, the volume calculated with additional integrators is within the error bounds from MC, but between 1 and 5 orders of magnitude faster. We attribute the speedup mainly to the reduction in number of MC samples. Thus, even for coarse volume estimates with tolerance 2.2×10^{-3}, we see the benefit of additional integrators.

6. CONCLUSIONS

We have developed a new generalized algorithm for computing volumes in CSG models. The algorithm follows a divide-and-conquer framework that intelligently applies specific integrators to each of the boxes of the octree that arise from the subdivision. We have described six different integrators, though the real strength of the algorithm is that it supports unlimited additional integrators. In this way, the algorithm can be specifically tailored to different problem types for decreased runtime and increased accuracy. The new algorithm is fully general and can handle any valid CSG component definition with up to second-order surfaces.

Table II. Numbers of boxes evaluated by different integrators for different models and tolerances, with volume and timing information provided. Cube model has no cylinders so the +Cyl integrator is omitted; other models have few planes so the +1 Plane integrator is omitted.

Model	Alg	Total	Integrators (% of total boxes)						Total	Time
Name		Boxes	MC	Box	1pl	2pl	Cyl	Bun	Volume	(sec)
Cube	MC	1	100.0	-	-	-	-	-	0.1245446	0.09
tol: ±2.2e-03	+octree	20,280	42.6	57.4	-	-	-	-	0.1251001	0.02
vol: 0.1249998	+1 Plane	4,768	15.4	58.6	26.0	-	-	-	0.1249996	<.01
	+2 Plane	610	3.0	64.3	9.8	23.0	-	-	0.1250019	<.01
Cube	MC	1	100.0	-	-	-	-	-	0.1249980	131.12
tol: ±1.1e-04	+octree	2,060,7294	42.9	57.1	-	-	-	-	0.1250000	18.16
vol: 0.1249998	+1 Plane	168,428	14.3	57.3	28.4	-	-	-	0.1249998	0.37
	+2 Plane	1,380	1.2	66.5	10.0	22.3	-	-	0.1249998	<.01
DrillCyl	MC	1	100.0	-	-	-	-	-	0.3867179	0.20
tol: ±2.2e-03	+octree	73,200	44.7	55.3	-	-	-	-	0.3866701	0.08
vol: 0.3866281	+2 Plane	66,144	43.2	55.1	1.3	0.4		-	0.3868198	0.08
	+Cyl	10,872	15.5	48.2	1.5	2.2	32.7	-	0.3866121	0.02
DrillCyl	MC	1	100.0	-	-	-	-	-	0.3866408	286.00
tol: ±1.1e-04	+octree	78,737,968	42.9	57.1	-	-	-	-	0.3866279	73.62
vol: 0.3866281	+2 Plane	67,367,700	42.8	57.1	<0.1	<0.1	-	-	0.3866278	62.73
	+Cyl	378,512	14.1	54.2	1.3	2.7	27.6	-	0.3866278	0.65
cPiped12	MC	1	100.0	-	-	-	-	-	0.0660653	0.13
tol: ±2.2e-03	+octree	24,753	51.4	48.6	-	-	-	-	0.0655403	0.03
vol: 0.0657512	+2 Plane	13,784	53.4	43.1	1.9	1.5	-	-	0.0658770	0.03
	+Cyl	8,667	32.7	39.3	2.5	2.4	23.1	-	0.0657460	0.02
	+Bun	3,322	22.3	44.5	6.3	6.2	12.9	8.1	0.0657594	0.02
cPiped12	MC	1	100.0	-	-	-	-	-	0.0657498	192.75
tol: ±1.1e-04	+octree	34,070,947	43.4	56.6	-	-	-	-	0.0657512	32.73
vol: 0.0657512	+2 Plane	20,543,405	43.7	56.2	<0.1	<0.1	-	-	0.0657509	20.05
	+Cyl	1,296,975	23.1	39.4	0.9	1.0	35.7	-	0.0657511	2.36
	+Bun	198,090	15.1	52.0	5.9	6.6	19.4	1.0	0.0657512	0.56
cPiped100	MC	1	100.0	-	-	-	-	-	0.0731463	0.60
tol: ±2.2e-03	+octree	23,003	62.4	37.6	-	-	-	-	0.0731258	0.07
vol: 0.0731920	+2 Plane	11,936	75.6	21.0	2.6	0.8	-	-	0.0733605	0.06
	+Cyl	11,887	68.2	20.7	2.6	0.8	7.6	-	0.0732219	0.06
	+Bun	3,228	28.7	34.8	7.8	3.1	2.3	24.0	0.0732155	0.03
cPiped100	MC	1	100.0	-	-	-	-	-	0.0731951	790.28
tol: ±1.1e-04	+octree	62,392,744	45.2	54.8	-	-	-	-	0.0731921	63.96
vol: 0.0731920	+2 Plane	48,958,575	45.6	54.2	<0.1	<0. I	-	-	0.0731919	51.32
	+Cyl	8,527,009	25.2	38.4	0.3	0.4	35.7	-	0.0731919	14.58
	+Bun	482,756	16.8	49.6	4.8	7.0	18.5	3.3	0.0731919	1.41
cPiped10000	MC	1	100.0	-	-	-	-	-	0.0768654	183.04
tol: ±2.2e-03	+octree	23,003	56.4	43.6	-	-	-	-	0.0767527	2.40
vol: 0.0767715	+2 Plane	11,936	63.4	32.4	3.3	0.8	-	-	0.0768464	2.33
	+Cyl	11,887	63.6	32.2	3.3	0.8	<0.1	-	0.0768258	2.34
	+Bun	3,228	25.0	39.5	9.6	3.1	0.3	23.1	0.0767881	2.36
cPiped10000	MC	-	-	-	-	-	-	-	-	N/A[a]
tol: ±1.1e-04	+octree	208,125,506	67.0	33.0	-	-	-	-	0.0767697	358.09
vol: 0.0767715	+2 Plane	195,211,080	68.6	31.4	<0.1	<0.1	-	-	0.0767696	348.25
	+Cyl	162,382,473	43.7	23.1	<0.1	<0.1	33.2	-	0.0767696	346.37
	+Bun	1,539,063	30.3	30.6	0.3	4.9	13.0	20.9	0.0767691	9.43

[a]Halted after 12 hours. Extrapolating from cPiped10000 with tolerance ±2.2e-3, time will be about 76 hours.

Table III. Percentages of the unit volumes integrated by the different integrators; the decrease in Monte Carlo volume is directly related to the number of samples and running time.

Model Name	Alg	Integrators (% of total volume) MC	Box	1pl	2pl	Cyl	Bun	Total Samples	Time (sec)
Cube	MC	100.0	-	-	-	-	-	999,995	0.09
tol: ±2.2e-03	+octree	3.3	96.7	-	-	-	-	34,528	0.02
vol: 0.1249998	+1 Plane	0.3	80.7	19.0	-	-	-	2,928	<.01
	+2 Plane	<0.1	61.1	9.3	29.6	-	-	72	<.01
Cube	MC	100.0	-	-	-	-	-	1,410,065,909	131.12
tol: ±1.1e-04	+octree	0.1	99.9	-	-	-	-	17,663,096	18.16
vol: 0.1249998	+1 Plane	<0.1	80.9	19.1	-	-	-	48,176	0.37
	+2 Plane	<0.1	61.1	9.3	29.6	-	-	32	<.01
DrillCyl	MC	100.0	-	-	-	-	-	999,995	0.20
tol: ±2.2e-03	+octree	12.5	87.5	-	-	-	-	130,784	0.08
vol: 0.3866281	+2 Plane	10.9	83.7	4.6	0.8	-	-	114,256	0.08
	+Cyl	0.6	36.1	2.5	0.8	60.0	-	6,720	0.02
DrillCyl	MC	100.0	-	-	-	-	-	1,410,065,909	286.00
tol: ±1.1e-04	+octree	0.4	99.6	-	-	-	-	67,494,688	73.62
vol: 0.3866281	+2 Plane	0.3	94.2	4.7	0.8	-	-	57,664,680	62.73
	+Cyl	<0.1	36.5	2.5	0.8	60.2	-	106,576	0.65
cPiped12	MC	100.0	-	-	-	-	-	999,995	0.13
tol: ±2.2e-03	+octree	4.9	95.1	-	-	-	-	50,932	0.03
vol: 0.0657512	+2 Plane	2.8	70.4	6.2	20.6	-	-	29,468	0.03
	+Cyl	1.1	69.2	6.2	20.6	2.9	-	11,332	0.02
	+Bun	0.3	68.3	6.2	20.6	0.9	3.8	2,964	0.02
cPiped12	MC	100.0	-	-	-	-	-	1,410,065,909	192.75
tol: ±1.1e-04	+octree	0.2	99.8	-	-	-	-	29,604,860	32.73
vol: 0.0657512	+2 Plane	0.1	73.0	6.2	20.6	-	-	17,953,588	20.05
	+Cyl	<0.1	69.8	6.2	20.6	3.4	-	598,522	2.36
	+Bun	<0.1	68.5	6.2	20.6	1.0	3.8	59,964	0.56
cPiped100	MC	100.0	-	-	-	-	-	999,995	0.60
tol: ±2.2e-03	+octree	5.5	94.5	-	-	-	-	57,392	0.07
vol: 0.0731920	+2 Plane	3.4	72.2	6.8	17.6	-	-	36,100	0.06
	+Cyl	3.1	72.2	6.8	17.6	0.4	-	32,440	0.06
	+Bun	0.4	70.4	6.7	17.6	<0.1	4.9	3,700	0.03
cPiped100	MC	100.0	-	-	-	-	-	1,410,065,909	790.28
tol: ±1.1e-04	+octree	0.3	99.7	-	-	-	-	56,352,288	63.96
vol: 0.0731920	+2 Plane	0.3	75.3	6.8	17.6	-	-	44,694,892	51.32
	+Cyl	<0.1	73.7	6.8	17.6	1.9	-	4,295,224	14.58
	+Bun	<0.1	70.5	6.7	17.6	0.1	5.0	162,002	1.41
cPiped10000	MC	100.0	-	-		-	-	999,995	183.04
tol: ±2.2e-03	+octree	5.0	95.0	-		-	-	51,920	2.40
vol: 0.0767715	+2 Plane	2.9	72.7	6.8	17.6	-	-	30,280	2.33
	+Cyl	2.9	72.7	6.8	17.6	<0.1	-	30,220	2.34
	+Bun	0.3	70.4	6.7	17.6	<0.1	4.9	3,232	2.36
cPiped10000	MC	-	-	-	-	-	-	-	N/A[a]
tol: ±1.1e-04	+octree	1.6	98.4	-	-	-	-	279,088,846	358.09
vol: 0.0767715	+2 Plane	1.6	74.0	6.8	17.6	-	-	267,848,220	348.25
	+Cyl	0.8	73.7	6.8	17.6	1.1	-	141,769,844	346.37
	+Bun	<0.1	70.5	6.7	17.6	<0.1	5.1	931,534	9.43

[a]Halted after 12 hours. Extrapolating from cPiped10000 with tolerance ±2.2e-3, time will be about 76 hours.

In addition, we believe that volume computation is just one of many possible applications for this algorithm. From our current structure it is easy to label boxes of the octree with components of the hierarchy that straddle them. Such a data structure may more quickly locate and track particles while solving the transport equation.

Our exploration of error bounds and their propagation is rudimentary at this point, and much more can be said about decisions that affect the error from the various integrators. For example, if we start with a bounding box whose minimal point's coordinates and side lengths are powers of two, each octree cell could have an exact representation. Thus, the volume of outside boxes would be exactly zero, and inside would be at most 3 ULP away from the true box volumes. Developing tight error bounds for the other integrators is not so easy. Perhaps by keeping tight bounds on the numerical errors introduced by an integrator we could make a more informed traversal of the octree.

ACKNOWLEDGMENTS

The authors would like to thank Ian Barefoot and Phillip Smith for their assistance in creating test models. Mr. Millman's research was performed under appointment to the Rickover Fellowship Program in Nuclear Engineering sponsored by Naval Reactors Division of the U.S. Department of Energy. Dr. Snoeyink acknowledges support from a contract with the Bettis Atomic Power Laboratory and from a NSF research grant.

REFERENCES

[1] T.M. Sutton, *et. al.*, “The MC21 Monte Carlo Transport Code,” *Joint International Topical Meeting on Mathematics and Computation, and Supercomputing in Nuclear Applications (M&C+SNA 2007)*, Monterey, California, April 15-19, 2007, on CD-ROM

[2] X-5 Monte Carlo Team, “MCNP – A General Monte Carlo N-Particle Transport Code Version 5,” *Technical Report LA-UR-03-1987*, Los Alamos National Laboratory Report (2003).

[3] A.G. Requicha, “Representations for Rigid Solids: Theory, Methods, and Systems,” *ACM Comput. Surv.*, **12**, pp. 437-464 (1980).

[4] H. Samet, “The Quadtree and Related Hierarchical Data Structures,” *ACM Comput Surv*, **16**, pp. 187-260 (1984).

[5] J. Levin, “A Parametric Algorithm for Drawing Pictures of Solid Objects Composed of Quadric Surfaces,” *Commun ACM*, **19** (10), pp. 555-563 (1976).

[6] E.B. Wilson, “Probable Inference, the Law of Succession, and Statistical Inference,” *Journal of the American Statistical Association*, **22** (158), pp. 209-212 (1927).