Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry

Tom Duff†

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Abstract

Recursive subdivision using interval arithmetic allows us to render CSG combinations of implicit function surfaces with or without anti-aliasing. Related algorithms will solve the collision detection problem for dynamic simulation, and allow us to compute mass, center of gravity, angular moments and other integral properties required for Newtonian dynamics.

Our hidden surface algorithms run in 'constant time.' Their running times are nearly independent of the number of primitives in a scene, for scenes in which the visible details are not much smaller than the pixels. The collision detection and integration algorithms are utterly robust — collisions are never missed due to numerical error and we can provide guaranteed bounds on the values of integrals.

CR Categories and Subject Descriptors: G.1.0 [Numerical Analysis] Numerical Algorithms I.3.3 [Picture and Image Generation] Display algorithms, Viewing algorithms, I.3.5 [Computational Geometry and Object Modeling] Curve, surface, solid and object representations, I.3.5 [Computational Geometry and Object Modeling] Hierarchy and geometric transformations, I.3.7 [Three-Dimensional Graphics and Realism] Visible line/surface algorithms, Animation

General Terms: Algorithms

Additional Keywords and Phrases: anti-aliasing, compositing, computer-aided animation, recursive subdivision, image synthesis, dynamic simulation, collision detection

1. Introduction

The most commonly-used geometric representations in computer graphics are local. Polygonal models, for example, specify which points are on an object's surface, and tell us nothing substantial about the rest of the space in which the object is embedded, except by omission. It requires substantial mental effort to formulate answers tp questions like "Do these objects intersect?", or "What parts of this object are visible?" or even something as simple as "What is the volume of this object?". More elaborate surface representations, like Bezier patches or NURBS don't make these questions any easier—since they only describe the objects locally, they make it difficult to answer global questions about them.

Likewise, the computational methods we normally use are mostly local. The ray-tracing algorithm, for example, tries to compute an image one pixel at a time by testing every primitive in the scene for intersection with a ray from the eye-point through the pixel's center. Of course any decent ray-tracer goes to a lot of trouble to avoid most of this work. But an algorithm that had decent access to global information about the scene wouldn't need go to the trouble—it would know immediately what parts of the scene were relevant to what parts of the screen.

†Phone (908) 582-6485, email td@research.att.com

A good example of a global representation is the BSP tree [11]. Each node of a BSP tree gives useful information about the object's relationship to the whole of the space it's embedded in. The nodes effectively say about their subtrees, "in this half of space, you need only think about this half of the model." BSP trees naturally engender simple algorithms for all sorts of geometric tasks, from hidden surface removal to object intersection [23] to shadow generation [6], that make natural, effective use of the global information stored in the model.

This paper will examine in detail another global object representation and its algorithms, based on implicit functions, Constructive Solid Geometry and interval arithmetic.

Briefly, implicit functions are test functions for classifying points in space as inside, on or outside an object. Interval arithmetic allows us to extend those tests to whole chunks of space at once. Constructive Solid Geometry allows us to combine simpler objects, keep unwieldy primitives (like infinite cylinders) under control and model many important industrial and natural processes that go into creating geometric forms.

2. Implicit Functions

Implicit functions are an indirect representation of solid objects. Given a function of three variables $F(x,y,z)$, we can use the equation $F(x,y,z)=0$ to specify the points on a surface. The representable surfaces range from the mundane to the exotic: from planes $(ax+by+cz+d=0)$ and quadrics—the spheres, cones cylinders and paraboloids of elementary geometry—to more exotic polynomial surfaces like those of Kummer and Dupin [10] to Barr's downright weird twisted, bent and tapered super-ellipsoids [4].

If F is continuous, we can classify points as inside, on or outside the object depending on whether $F<0$, $F=0$ or $F>0$. This is the global property we are after: F classifies every point in space in its relationship to the surface. In regions of space not crossed by the surface, the fact that F's sign does not change is a source of coherence useful in hidden-surface and other geometric algorithms that can be exploited by using interval arithmetic to quickly obtain bounds on $F(x,y,z)$ for whole ranges of x, y and z.

3. Interval Arithmetic

Interval arithmetic [16] generalizes ordinary arithmetic to closed, bounded ranges of real numbers. If $\underline{X}$ and $\overline{X}$ are real numbers with $\underline{X}\le\overline{X}$, then X is an interval

$$X=[\underline{X},\overline{X}]=\{x\,|\,\underline{X}\le x\le\overline{X}\}$$

 ACM-0-89791-479-1/92/007/0131 $01.50

The natural interval extensions of the elementary operations of arithmetic are

$$\begin{aligned} X+Y &= [\underline{X}+\underline{Y}, \overline{X}+\overline{Y}] \\ X-Y &= [\underline{X}-\overline{Y}, \overline{X}-\underline{Y}] \\ X\times Y &= [\min(\underline{X}\underline{Y}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{X}\overline{Y}), \max(\underline{X}\underline{Y}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{X}\overline{Y})] \\ X/Y &= [1/\overline{Y}, 1/\underline{Y}], \text{ but undefined if } \underline{Y} \le 0 \le \overline{Y} \end{aligned} \quad (1)$$

These definitions give tight bounds on the range of the corresponding real functions with arguments chosen in the given intervals. In particular, for degenerate intervals like $[a,a]$, interval arithmetic reproduces ordinary arithmetic. We can use these rules to compute bounds on the value of a rational expression $F(x,y,z)$ inside any box (X,Y,Z).

Unfortunately, the achievable bounds on general arithmetic expressions are not as tight as on the arithmetic operators. For example, $x^2 \in [0,1]$ when $x \in [-1,1]$, but, by (1), $x \times x = [-1,1]$. Generally, for intervals X, Y, Z,

$$\{F(x,y,z) \mid x \in X, y \in Y, z \in Z\} \subseteq F(X,Y,Z)$$

but we cannot replace $\subseteq$ by $=$ except in special circumstances.

A second source of looseness is that when using finite precision floating-point arithmetic, we must be sure to round the upper and lower bounds in the appropriate directions. Doing this is not the practical problem that it used to be — machines claiming to do IEEE arithmetic [12] are required to provide control of the rounding direction of floating-point calculations. If instead you use improperly-rounded interval arithmetic, the errors introduced will not often be noticeable. Of course, such an implementation voids the warranty of robustness.

Any $F(X,Y,Z)$ composed using the rules (1) above is an *interval extension* of the corresponding real function. That is, $F([x,x],[y,y],[z,z]) = [F(x,y,z), F(x,y,z)]$. Furthermore, F is *inclusion monotonic*. That is, if $X' \subseteq X$, $Y' \subseteq Y$ and $Z' \subseteq Z$, then $F(X',Y',Z') \subseteq F(X,Y,Z)$. Moore [16] is a good general introduction to interval methods, and discusses these properties and their implications in detail. For purposes of this paper, it is sufficient that if $x \in X$, $y \in Y$ and $z \in Z$ that $F(x,y,z) \in F(X,Y,Z)$. This is true for every inclusion monotonic interval extension of $F(x,y,z)$.

We can easily construct interval extensions of most standard transcendental functions. For monotonic functions like e^x, $\ln x$, $\sqrt{x}$, we have $F(X) = [F(\underline{X}), F(\overline{X})]$, or if $F(x)$ is monotone decreasing, $F(X) = [F(\overline{X}), F(\underline{X})]$. Continuous functions that have maxima and minima in known places, like sin and cos, can be handled by taking the union of their values over monotonic pieces. For example

$$X^n = \begin{cases} [\underline{X}^n, \overline{X}^n] & n \text{ odd or } \underline{X} \ge 0 \\ [\overline{X}^n, \underline{X}^n] & n \text{ even and } \overline{X} \le 0 \\ [0, \max(-\underline{X}, \overline{X})^n] & n \text{ even and } \underline{X} < 0 < \overline{X} \end{cases}$$

These interval extensions all give tight bounds on the underlying transcendental functions, and expressions involving them yield inclusion monotonic interval extensions of the underlying real expressions.

If $\overline{F}(X,Y,Z) < 0$, we know that all points (x,y,z) with $x \in X$, $y \in Y$ and $z \in Z$ are located inside the implicit function surface F, and if $\underline{F}(X,Y,Z) > 0$ they are all outside. If $\underline{F}(X,Y,Z) \le 0 \le \overline{F}(X,Y,Z)$, we can guess that the surface might intersect the cell (we cannot be sure unless we know that the bounds we've computed are tight) and that it deserves closer examination.

4. Constructive Solid Geometry

A powerful and natural tool for taming implicit functions and building useful geometric models from them is Constructive Solid Geometry (CSG). Since implicit functions describe volumes as point-sets, we can use them as primitives and build more complicated models using set-theoretic union, intersection, complement and difference operators. The union and set-difference operators can model the most important ways that people build real objects. Milling machines, saws, drills, routers and chisels are all (restricted) set-difference engines. Glue, nails, soldering irons and Velcro are set-union agents—in the world of real solids, all unions are of disjoint sets. Set intersection allows us to focus attention on interesting or useful local features of implicit functions that may extend to infinity or otherwise behave wildly at a distance.

In general, we will represent an object or scene as a tree with implicit functions at its leaves and CSG operators at its interior nodes. We will assume that the only operators in the tree are union ($S \cup T$) and intersection ($S \cap T$). Set difference ($S - T$) and complement ($\neg S$) operators can be eliminated by repeatedly applying the rules

$$\begin{aligned} &S - T \to S \cap \neg T \\ &\neg(S \cup T) \to \neg S \cap \neg T \\ &\neg(S \cap T) \to \neg S \cup \neg T \\ &\neg F \to -F, \text{ where } F \text{ is a leaf function} \end{aligned}$$

The first rule converts set differences into intersections. The second two (deMorgan's laws) push complement operators toward the leaves. The third absorbs complement operators into the leaf functions.

5. Rendering

Suppose we wish to make shaded images of a scene described as a CSG combination of implicit-function primitives. For simplicity, we will make an image by parallel projection in the z direction into the xy plane. (Perspective is a simple extension — we can either incorporate the viewing transformation directly into the CSG tree's leaf functions or decorate the CSG tree with transformation matrices and transform coordinates as we walk the tree.)

We are given a CSG tree and a rectangular viewing volume described by three intervals (X,Y,Z). For each leaf of the tree, we can do an interval computation to bound the value of the leaf's function in the viewing cell. If the upper bound is negative or the lower bound is positive, we can replace the leaf by the empty set $\varnothing$ or its complement U and simplify the tree by repeatedly applying the rules

$$\begin{aligned} &\varnothing \cap S \to \varnothing, \quad U \cap S \to S \\ &\varnothing \cup S \to S, \quad U \cup S \to U \end{aligned}$$

Now we can divide the viewing cell into 8 pieces by dividing X, Y and Z at their midpoints and repeat this procedure recursively. At each level of subdivision, replacement of primitives by constants will further reduce the CSG tree—if we are lucky, the whole tree will reduce to a constant, in which case we need consider the cell no further, since it contains no surface. (We should always keep in mind that the reduced CSG trees are valid only within the corresponding cells.) Subdivision terminates when the bases of the cells are pixel-sized, at which point the reduced CSG tree should be very small—typically only the one or two primitives contributing to the image at the pixel.

At each level of subdivision we should first examine the sub-cells closest to the viewpoint and use a quad-tree or some equivalent data structure to keep track of which pixels are

completely covered, allowing us easily to avoid examining more distant cells that will contribute nothing to the image. (This algorithm follows directly from the *vole* algorithm, described by Woodwark and Quinlan [25]. They use a similar subdivision scheme, but for CSG models whose only primitives are planar half-spaces, for which they need not resort to interval arithmetic to classify cells.)

The total expenditure required to sample the surface is nearly independent of the number of primitives in the model. This obviously cannot be true in the limit. Let us call a model "realistic" if by-and-large no more than a few primitive surfaces cross each pixel. (On a 1000×1000 screen, a model can have several million (small) primitives and still satisfy this requirement.) For realistic models, small cells will by-and-large contain reduced CSG trees with only a few nodes. Large cells, containing more complex trees, are *much* fewer in number — depending on how many cells are culled by the coverage quad-tree, there are between three and seven times as many pixel-sized cells as there are cells of all other sizes combined. The great majority of the computation occurs in small cells, in which the size of the reduced CSG trees does not strongly depend on the complexity of the original model.

When we have subdivided to the pixel level, we can sample the image using ordinary ray-casting methods. We substitute the ray's parametric equation $(x,y,z)=A+\alpha(B-A)$, where A and B are appropriate points on the near and far planes of the subdivided cell, into each primitive of the reduced CSG tree and find all values of α where the primitives go to zero. We discard values of α for which the corresponding point is outside the object, as determined by substituting point coordinates into the leaves and evaluating the CSG operators. The smallest remaining α, if any, denotes the visible point on the surface. Papers by Amanatides and Mitchell ([14], [15], [1]) answer in much more detail questions that may arise in implementing this sort of ray-casting procedure.

6. Anti-aliasing

If point-sampling dissatisfies you (as it ought to!) then interval arithmetic can help do better. If we ignore for now the problems of highlight and texture aliasing (see [7], among numerous others, for apposite approaches to these aspects of the problem), then the anti-aliasing question hinges on identifying silhouette edges of primitives and intersection edges of CSG combinations.

Ideally, we would compute in their entirety the visible portions of all significant edges and use an exact convolution method like that of [9] to compute an anti-aliased image. As this appears to be too much to hope for, we must satisfy ourselves with a careful treatment of the simple cases that affect most pixels, approximating the rest as well as we can afford. (For simplicity, we will assume that we are using a box filter to sample the image although that is by no means a limitation of the method—see [9] for more relevant discussion. That is, we compute pixel values by integrating the scene's intensity across pixel-sized squares.)

As above, we will subdivide the viewing volume, reducing CSG trees as we go. When we reach pixel resolution, we will concentrate our best attention on cells that are completely covered by one primitive (figure 1a) or are crossed by a single visible edge. This edge will be either the silhouette edge of a tree with a single leaf (figure 1b) or an intersection edge in a two-leaf tree (figure 1c). We will treat more complicated situations (figure 1d) by subdividing the pixel, hoping to find a simpler situation in the subpixels and giving up when their contribution to the image is tiny.

Suppose that each implicit function F at the leaves of our CSG tree has continuous partial derivatives. The silhouette of the surface $F=0$ is precisely those points at which $\partial F/\partial z=0$. So, let us use interval arithmetic to evaluate $\partial F/\partial z$ in our pixel-sized cell, calling the result S. If $0\notin S$, the surface has no edge inside the cell. It may, however, protrude through the cell's front or back surface (figure 1e), giving a spurious edge crossing the pixel where the surface is clipped. An interval computation can alert us to this possibility. If $0\in F(X,Y,[\bar{Z},\bar{Z}])$ then F may pass through the far surface of the cell (X,Y,Z). Whenever we come to process a cell that contains such a surface, we save the CSG tree and the cell coordinates and defer processing the cell. When later we return to the cell behind it, we can merge the two cells and their trees before processing them. (Of course, if the CSG tree in the cell behind has been reduced to $\varnothing$ or U, we will not return to it. When later we return to the pixel for farther cells, we should first dispose of the saved tree by treating it as one of the more complicated cases mentioned above. You might, as I did initially, naively believe that this circumstance cannot occur — after all, if there is nothing in the cell behind, how could a surface cross the cell boundary? However, if one surface of an intersection passes through the back of a cell and the other does not, the cell behind may easily reduce to $\varnothing$.)

Now we are ready to handle the simple cases:

If the reduced CSG tree has only one leaf and $0\notin S$, the primitive has no edges inside the cell (figure 1a) and may trivially be rendered by casting a ray through the pixel's center.

If the CSG tree has two leaves and $0\notin S$ for each, then we have two possibly intersecting primitives F and G, neither with an edge in the cell (figure 1c). We will approximate their intersection curve by a straight line that runs from edge to edge of the pixel. We can cast four rays to find the Z coordinates at which the surfaces pass through the pixel corners. Then, following [8], we compare the z values at each corner. Along each pixel edge where the z's compare differently we linearly interpolate the z's to find a point at which they are equal. In any cell there will be zero, two or four such points. We join them up as in figure 2 (taken from [8]) and compute the quantity β, the total fraction of the pixel in which F is in front of G, according to our approximation. Now we need only determine which surface, if any, is visible in each part of the cell and compute their contributions, weighted by β, to the color of the cell. We can determine visibility by considering the CSG operator connecting the two surfaces, and whether the two surfaces face the viewer. For example, if F faces the viewer and G faces away, and the CSG operator is $\cap$, then we can see F inside the part of the pixel where it is in front, and nothing in the other part. The following table summarizes the contributions in all cases:

	F toward G toward	F toward G away	F away G toward	F away G away
$\cup$	$\beta F+(1-\beta)G$	$(1-\beta)G$	βF	$\beta G+(1-\beta)F$
$\cap$	$\beta G+(1-\beta)F$	βF	$(1-\beta)G$	$\beta F+(1-\beta)G$

If the reduced CSG tree has only one leaf and $0\in S$, then there may be a silhouette edge in the cell, as in figure 1b. As before, we wish to approximate the edge by a segment running from edge to edge of the pixel. Again, we can cast rays through the pixel corners, but now we are interested only in identifying cell faces through which the edge must pass because the surface intersects one edge of the face but not the other. Having identified these faces (again there must be zero, two or four of them), we can find the silhouette's endpoints by solving systems

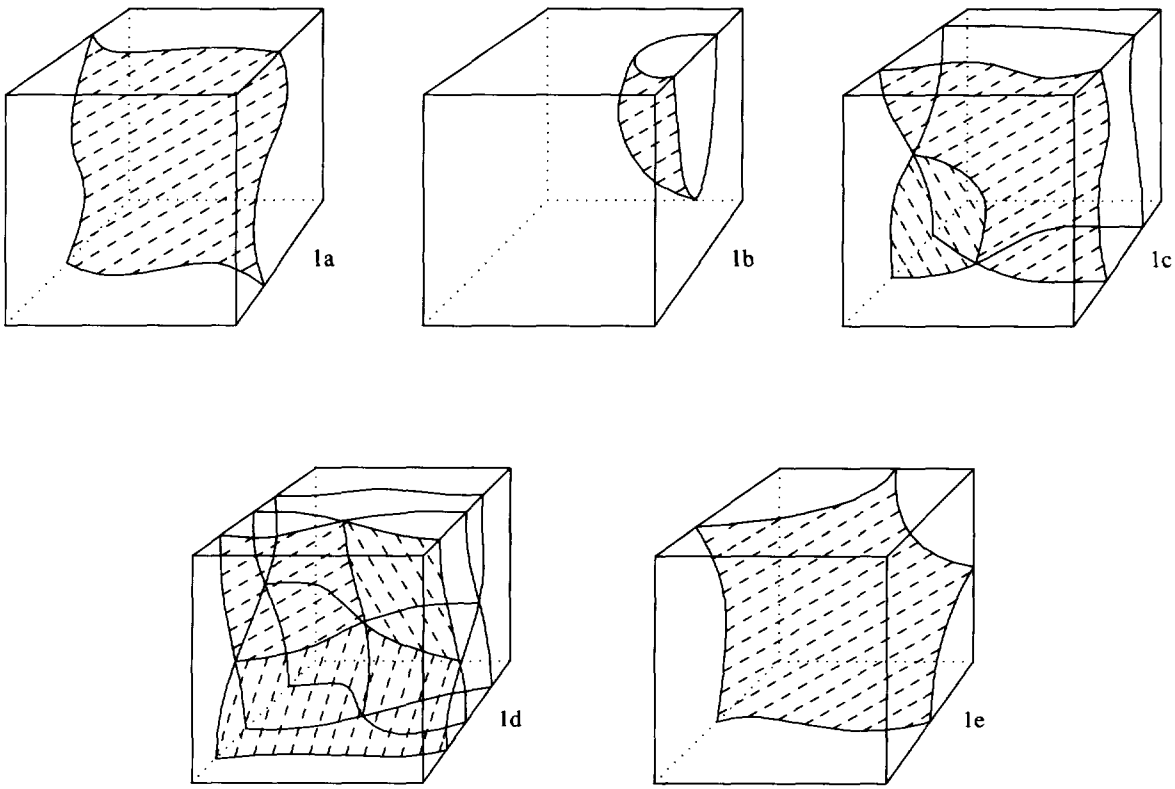

Figure 1 – Simple and non-simple cells
Figure 1a illustrates a cell completely covered by a single surface. 1b has a cell containing a single surface with a silhouette edge. 1c has a cell covered by a pair of intersecting surfaces. 1d is too complicated to be handled without subdivision. 1e illustrates a cell with a surface passing through its back face. Its processing will be deferred for merging with the cell behind it.

of three simultaneous equations: $F(x,y,z)=0$, $\partial F/\partial z=0$ and the plane equation of the cell face, one of $x=\underline{X}$, $x=\overline{X}$, $y=\underline{Y}$ or $y=\overline{Y}$. A useful optimization is to eliminate the cell face equation by substituting into the other two. Moore ([16] pp 62-68) outlines robust interval methods for producing these solutions.

As in the previous case, we can join these points by line segments (see figure 3) to find the fraction of the pixel covered by the surface. Note that this computation is not particularly robust — it is entirely possible for an edge to protrude a pixel without the surface passing through any vertex. Indeed, the whole surface may be contained within a single cell. But, we stated up front that we intended to approximate the silhouette by a sequence of line segments passing from pixel edge to pixel edge, and neither of these situations admits a reasonable approximation in those terms. A more robust computation that could identify these situations

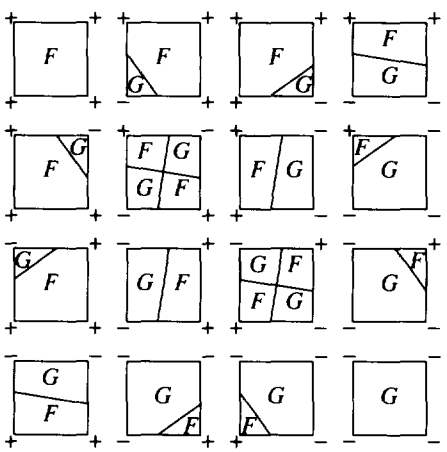

Figure 2 – Joining endpoints of edges
Each square represents a pixel. The corners are marked with the sign of $F-G$. The label in each pixel fragment indicates which surface is in front in it. β is the total area of the fragments labeled F.

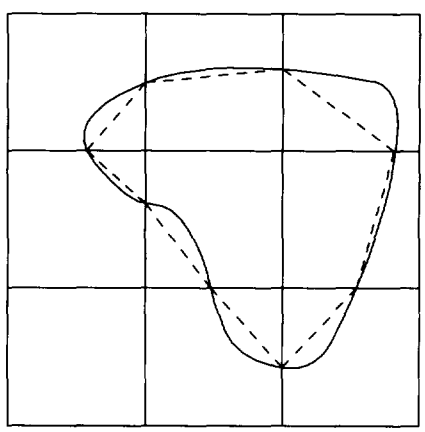

Figure 3 – silhouette edge approximation
The squares are pixels. The closed curve is the a silhouette edge of a surface that intersects each pixel as in figure 1a. The dashed lines are our piecewise linear approximation to the silhouette.

and allow us to subdivide would be straightforward, if tedious, to implement.

Any situation not handled above we manage by further subpixel subdivision. We subdivide only in x and y, not z, further reducing the CSG tree as we go, stopping when we find a subpixel satisfying one of the above cases or in any case at some fixed depth. The contributions of the subpixel cells, weighted by subpixel area, are added to determine the color of their pixel-sized ancestor.

Since cells that partly cover pixels must allow the color of cells behind them to show through the uncovered parts, we store colors in the rgbα representation of Porter and Duff [19], using α-blending to composite each cell's color with the accumulated color of the cells in front of it. This can make mistakes in pixels that contain multiple visible edges. If this worries you, you can use a more elaborate compositing scheme, saving a list of previously-encountered edges against which to clip the newly-computed regions or keeping at each pixel a sub-pixel bit-mask (as in Carpenter's A-buffer [5], but simpler because we know depth order *a priori*) against which to clip the newly-computed regions.

7. Collision detection

Let us now turn our attention to motion computations for animation purposes. Suppose we have a scene composed of a number of CSG objects, and that we wish to compute their motion. At any point in time, we need to decide whether objects in the scene have collided, in order to prevent interpenetration and to compute the forces resulting from any collision.

Let the objects in the scene (considered as point-sets) be O_i, $0 \le i < n$. Then to compute an image we would run one of the above algorithms on the CSG tree for

$$\bigcup_{i=0}^{n-1} O_i \tag{2}$$

To decide whether there has been a collision, we must decide whether there are points occupied by more than one object. To do this, we first build a CSG tree in which each of the top-level union operators (see equation 2) is specially marked. Now, we recursively subdivide a cell surrounding the entire scene, reducing the CSG tree at each subcell as before, except that for specially-marked top-level union operators we rewrite the tree using only the rule

$$\varnothing \cup S \rightarrow S$$

We do not reduce top-level instances of U using the rule $U \cup S \rightarrow U$ because we wish to count them. If at any level of subdivision the reduced CSG tree contains two or more top-level U's, we have detected a collision. Likewise, if the tree has no marked top-level union operator, there can be no collision in this cell and we need subdivide no further.

If we subdivide down to cells smaller than some tolerance without reaching a decision, we can declare the question unanswerable to within the given tolerance. This may strike you as unsatisfactory, and in fact we can make a much stronger statement. For all functions computed by sequences of arithmetic operations and transcendental functions, their natural interval extensions satisfy an *interval Lipschitz condition* (under certain mild assumptions.) That is, there is an easy-to-compute constant c, depending only on F and the domain of interest (say a viewing volume (X_0,Y_0,Z_0)), such that if $(X,Y,Z) \subseteq (X_0,Y_0,Z_0)$ then $\bar{F}(X,Y,Z) - \underline{F}(X,Y,Z) \le c \max(\bar{X}-\underline{X}, \bar{Y}-\underline{Y}, \bar{Z}-\underline{Z})$. (Again, we refer you to [16] pp 33-35 for further details, including a proof.) Informally, the size of the intervals $F(X,Y,Z)$ decreases at worst linearly with the size of (X,Y,Z). Thus, when we abandon subdivision at a tolerance ε, we know that the surfaces touch to within some easily-computable tolerance $c\varepsilon$.

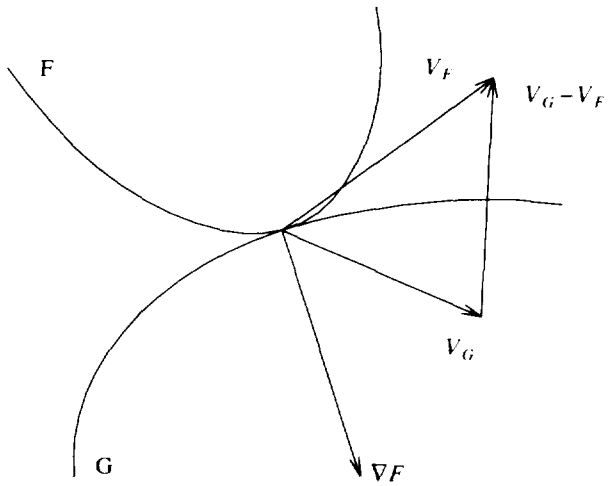

Figure 4 — collision
F and G are two colliding surfaces. ∇F is the normal to F at the point of collision. V_F and V_G are their velocities at the point of collision. Since the angle between ∇F and $V_G - V_F$ is larger than 90 degrees, surfaces are separating and the collision will be rejected.

8. Dynamic Collision Detection

If we can describe the motion of O_i as a function of time, we can extend the above algorithm to find the earliest time in an animation at which a collision occurs. We need only subdivide in x, y, z and t, looking at subcells earlier in time before later cells and stopping when we first find a cell smaller than a collision tolerance ε that still contains parts of two intersecting objects.

At this point we will presumably do a momentum-transfer calculation and restart the collision detector with $t = t_{collision}$. But this will just re-detect the previous collision. We must reject collisions at which the objects are not approaching one another. Let the two objects be F and G, and let their velocities at the point of collision by V_F and V_G — see figure 4. G is moving towards F if the angle between its velocity (relative to F) and F's normal at the point of collision is larger than 90 degrees, that is if $\nabla F \cdot (V_G - V_F) < 0$. In an edge-to-edge or other complicated collision, F's gradient may be undefined. In that case, we must use the normal to the collision tangent plane as described by Baraff [3].

The sorts of motion that this scheme can accommodate are fairly general. Any time-varying coordinate mapping will do, as long as its inverse can be expressed in a closed form that admits an inclusion monotonic interval extension. For example, points on a rigid body tumbling and moving under gravity are transformed by

$$P = \mathrm{rot}(\omega t, A) R(P' - C) + C + P_0 + V_0 t + \begin{bmatrix} 0 \\ -1/2 g t^2 \\ 0 \end{bmatrix} \tag{3}$$

where

$P = (x,y,z)^T$ is the transformed point,
$\mathrm{rot}(\theta, axis)$ is a rotation matrix,
ω is the object's rate of rotation,
A is the axis about which it rotates,

R is a rotation matrix describing its orientation at $t=0$,
P' is a coordinate in model-definition space,
C is the object's center of mass,
P_0 is its position at time $t=0$,
V_0 is its velocity at time $t=0$, and
g is the acceleration due to gravity.

The inverse of (3) is just

$$P' = R^T \mathrm{rot}(-\omega t, a)(P - C - P_0 - V_0 t - \begin{bmatrix} 0 \\ -1/2 g t^2 \\ 0 \end{bmatrix}) + C$$

So, the implicit function $F(P')$ is just F as transformed by its motion.

More complex motions, like the modal deformations of Pentland and Williams [18] can be handled similarly. Constrained motions like those described by Barzel and Barr [2] and Baraff [3] for which the associated ODEs are generally insoluble in closed form are beyond the scope of the work reported here. Interval methods for ODEs are an interesting research problem and would be extremely helpful here. In their absence, we must use conventional ODE methods and accept the loss of robustness that they entail.

Pentland and Williams [18] claim to do collision detection of implicit functions (but not their CSG combinations) by converting one of a pair of objects to be tested into polygons. The objects intersect if any of the vertices gives a negative value when substituted into the other function. Sclaroff and Pentland [20] repeat this claim. But, their scheme does not work—it is easy for an object to pass through a polygonal face without meeting any of its edges. Even testing the polygonal representation of each object against the other will not work, as they can easily meet edge-to-edge with no vertex of either polygonization penetrating the other object. The methods presented here are utterly robust—interval arithmetic always provides guaranteed bounds on the functions we compute.

When we discover that two objects meet, we need to calculate the collision forces and their effects on the bodies' motions. To do this, we need to know in what direction the force is applied and some physical properties of the colliding bodies—particularly their masses and moments of inertia.

The information needed to calculate the direction of applied force is readily available when the collision is detected. Inside the cell in which the collision occurs the reduced CSG trees will include one or more surfaces from each of the colliding objects. If a single surface from one object or the other is involved, we need only compute its normal direction. If each object has two surfaces active, we have an edge-to-edge collision. We can find the edge directions by looking at intersections of tangent planes, and transmit the force as in [3]. More complicated situations represent indeterminate cases that can also be linearized by working with the tangent planes and handled as in [3].

9. Integral Properties

Mass and moments of inertia are *integral properties* of solid objects. Computing them involves evaluating simple definite integrals inside the objects' volumes. For example, to compute the mass of a body B, we need to evaluate

$$\iiint_B \rho(x,y,z)\,dx\,dy\,dz$$

where ρ is the density of the material. If ρ is easy to integrate over rectangular prisms (often it will be constant), we can recursively subdivide a cell surrounding B, reducing B's CSG tree as we go, and accumulate the integral's value over those cells in which the reduced CSG tree is U. Cells whose reduced CSG trees are $\varnothing$ contribute nothing, and partially occupied cells will contain a vanishingly small fraction of B's volume as the subdivision limit decreases. (The fraction may not decrease as quickly as you'd like if the Hausdorff dimension of B's surface is larger than 2, but then you have worse problems since, for example, B's partial derivatives will be undefined.)

Other integral properties can be computed similarly. For example, B's moment of inertia about a particular axis is just

$$\iiint_B r^2 \rho(x,y,z)\,dx\,dy\,dz$$

where r is the distance from (x,y,z) to the axis in question.

10. Examples

Figures 5-9 show a variety of objects as rendered by our algorithms. Figure 5 is Kummer's surface with 16 real double-points (4 are at infinity) with malachite texture. This beautiful surface extends to infinity in 8 directions and would be useless for real applications without some sort of trimming. Figure 6 is the intersection of Kummer's surface and a sphere. Figure 7 is the intersection of a Parabolic Spindle Cyclide and a sphere with hideous orange marble texture. (Fischer [10] gives good detailed descriptions of these surfaces.) Figure 8 is an image of a face made using a 48×48 raster of intersecting marbles of varying sizes, just to show that we can handle scenes with a larger number of primitives. Figures 5-8 were all computed at 1024×1024 resolution using the point-sampling renderer described in section 5. Figure 9 is an anti-aliased rendering of a compound of 3 spheres, done at 256×256 resolution using the algorithm of section 6.

The videotape accompanying this paper shows two simple animations made using our dynamic collision detection and rendering algorithms. The first scene shows 9 balls falling onto a sphere with a dish carved out of its top. You can see the balls collide with the dish and, in one case, with each other. The second scene is similar, but with 25 balls. There are 80 collisions in this shot, mostly between pairs of balls.

11. Conclusion

We have presented a wide range of algorithms that use interval arithmetic and recursive subdivision of object space to process geometric objects described as CSG combinations of implicit function primitives.

The algorithms are all suited to manipulating extremely complex objects because they discard parts of the objects that are irrelevant to the subdivided cells. Their running times are only mildly influenced by the size of their inputs because small cells typically contain at most one or two surfaces. Presumably when the number of primitives in the original model is a large fraction of the number of pixels on the screen we will start to see greater dependence, as this assumption will begin to break down.

The algorithms of sections 7, 8 and 9 are quite robust—it is impossible to lose track of parts of objects due to rounding error when using interval arithmetic. The bounds it provides are absolutely guaranteed to enclose the exact function values. We can only run into trouble when we terminate subdivision at some *a priori* level, and even then the existence of interval Lipschitz conditions can help us set that level to bound the unavoidable error in our computations however we wish.

In retrospect, the work most closely related to ours is work by Al Barr and his colleagues on rendering and collision detection of functions with Lipschitz conditions [13], [24]. A Lipschitz

condition (not to be confused with an interval Lipschitz condition) is a bound on a function's variation. Given an appropriate Lipschitz constant, one can easily bound a function's value on an interval, a sort of "interval arithmetic without the intervals." In fact, Lipschitz constants can be computed by interval evaluation of a function's derivatives, and those bounds converted into bounds on the original function using the mean value theorem. In this light it is perplexing that Kalra and Barr [13] put the question of 'identifying ... useful implicit functions and computing Lipshitz constants' for them first on their list of important problems to attack.

Recursive subdivision using interval arithmetic is a natural and versatile scheme to use for implicit function CSG models. We have only begun to scratch the surface of its potential applications. Our anti-aliased rendering method should be easily convertible into a polygonization algorithm. (Indeed, Snyder [22] gives an interval polygonization algorithm, along with many other applications of interval arithmetic.) Interval function minimization methods can provide global optima for many problems and should be applicable to some of the control problems in animation, and, as Don Mitchell has pointed out in conversation, to a range of global illumination problems as well.

Another problem that must be better addressed before we can consider wider use of implicit function surfaces is the problem of using them to model sculpted surfaces, a realm in which Bezier surfaces and NURBS reign. There is some hope that this situation will improve. [17] and [21] are two recent papers describing ideas that show a great deal of promise.

12. Acknowledgements

Don Mitchell is always a good friend and source of ideas and criticism. His paper on interval root-finding [14] put this bee in my bonnet, and his interval arithmetic routines made it possible to get the first version of this stuff going in a couple of days.

Andy Witkin suggested looking at collision detection.

A detailed and insightful referee's report contributed greatly to the paper's clarity and correctness.

13. References

[1] John Amanatides and Don P. Mitchell, "Some Regularization Problems in Ray Tracing," *Proc. Graphics Interface '90*, 1990

[2] Ronen Barzel and Alan H. Barr, "A Modeling System Based on Dynamic Constraints," *Computer Graphics* 22(3), July 1988, 179-188

[3] David Baraff, "Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies," *Computer Graphics* 23(3), July 1989, 223-231

[4] Alan H. Barr, "Global and Local Deformations of Solid Primitives," *Computer Graphics* 18(3), July 1984, 21-30

[5] Loren Carpenter, "The A-Buffer, An Anti-Aliased Hidden Surface Method," *Computer Graphics* 18(3), July 1984, 103-108

[6] Norman Chin and Steven Feiner, "Near Real-Time Shadow Generation Using BSP Trees," *Computer Graphics* 23(3), July 1989, 99-106

[7] Franklin C. Crow, "Summed Area Tables for Texture Mapping," *Computer Graphics* 18(3), July 1984, 207-212

[8] Tom Duff, "Compositing 3-D Rendered Images," *Computer Graphics* 19(3), July 1985, 270-275

[9] Tom Duff, "Polygon Scan Conversion by Exact Convolution," *Raster Imaging and Digital Typography '89*, Cambridge University Press, London, 1989

[10] Gerd Fischer, *Mathematische Modelle/Mathematical Models*, Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden, 1986

[11] H. Fuchs, Z. M. Kedem and B. F. Naylor, "On Visible Surface Generation by *A Priori* Tree Structures," *Computer Graphics* 14(3), July 1980, 124-133

[12] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, New York, 1985

[13] Devandra Kalra and Alan H. Barr, "Guaranteed Ray Intersections with Implicit Surfaces," *Computer Graphics* 23(3), July 1989, 297-306

[14] Don P. Mitchell, "Robust Ray Intersection with Interval Arithmetic," *Proc. Graphics Interface '90*, 1990

[15] Don P. Mitchell, "Spectrally Optimal Sampling for Distribution Ray Tracing," *Computer Graphics* 25(3), July 1991, 157-164

[16] Ramon E. Moore, *Methods and Applications of Interval Analysis*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1979

[17] Shugeru Muraki, "Volumetric Shape Description of Range Data using "Blobby Model"," *Computer Graphics* 25(4), July 1991, 227-235

[18] Alex Pentland and John Williams, "Good Vibrations: Model Dynamics for Graphics and Animation," *Computer Graphics* 23(3), July 1989, 215-222

[19] Thomas Porter and Tom Duff, "Compositing Digital Images," *Computer Graphics* 18(3), July 1984, 253-259

[20] Sclaroff and Alex Pentland, "Generalized Implicit Functions for Computer Graphics," *Computer Graphics* 25(4), July 1991, 247-250

[21] Thomas W. Sederberg and Alan K. Zundel, "Scan Line Display of Algebraic Surfaces," *Computer Graphics* 23(3), July 1989, 147-156

[22] John Snyder, *Generative Modeling: An Approach to High Level Shape Design for Computer Graphics and CAD*, Ph.D. Thesis, California Institute of Technology, 1991

[23] W. Thibault and B. F. Naylor, "Set Operations on Polyhedra Using Binary Space Partitioning Trees," *Computer Graphics* 21(4), July 1987, 153-162

[24] Brian von Herzen, Alan H. Barr and Harold R. Zatz, "Geometric Collisions for Time-Dependent Parametric Surfaces," *Computer Graphics* 24(4), August 1990, 39-48

[25] J. R. Woodwark and K. M. Quinlan, "Reducing the effect of complexity on volume model evaluation," *Computer-Aided Design* 14(2), March 1982, 89-95

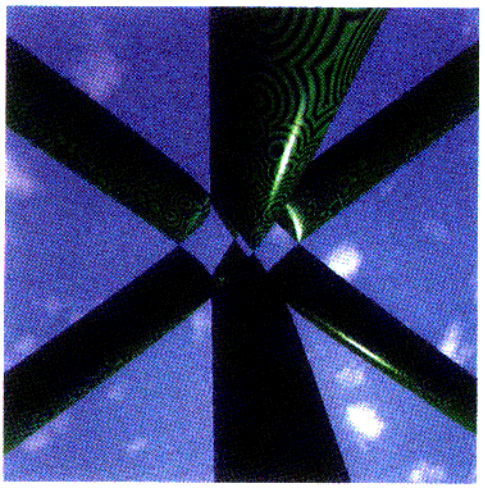

Figure 5 — Kummer surface

Figure 6 — Sphere-Kummer intersection

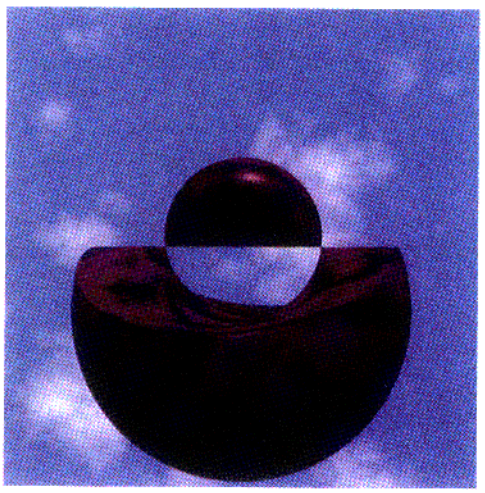

Figure 7 — Sphere-Cyclide intersection

Figure 8 — 2304 Spheres

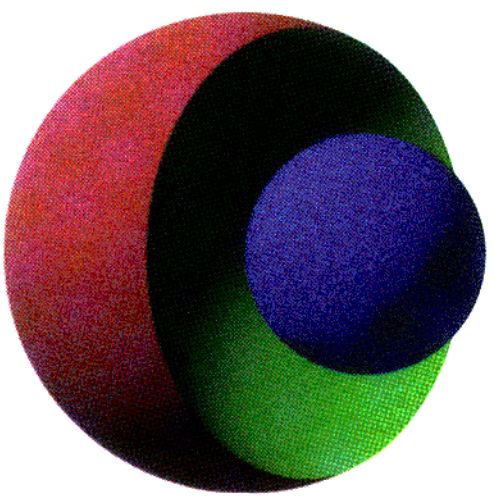

Figure 9 — Combination of 3 spheres, anti-aliased