Bounding Recursive Procedural Models using Convex Optimization

Orion Sky Lawlor and John C. Hart
Department of Computer Science
University of Illinois at Urbana-Champaign
olawlor@acm.org, jch@cs.uiuc.edu

Abstract

We present an algorithm to construct a tight bounding polyhedron for a recursive procedural model. We first use an iterated function system (IFS) to represent the extent of the procedural model. Then we present a novel algorithm that expresses the IFS-bounding problem as a set of linear constraints on a linear objective function, which can then be solved via standard techniques for linear convex optimization. As such, our algorithm is guaranteed to find the recursively optimal bounding polyhedron, if it exists. Finally, we demonstrate examples of this algorithm on two and three dimensional recursive procedural models.

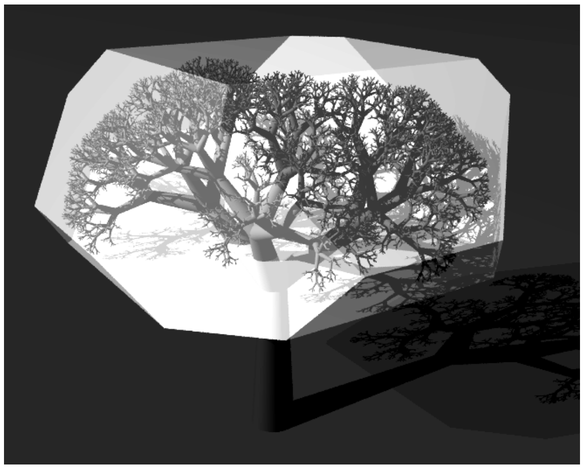

Figure 1. A truncated cube bounding volume, computed automatically by our algorithm to bound the leaves of a fractal tree, by bounding the tree's IFS.

1 Introduction

Recursive procedural models are an essential tool in computer graphics [8]. They allow a designer to generate scenes of any desired detail by specifying the rules or parameters to procedurally synthesize the scene instead of meticulously modeling every individual feature.

Procedural models are most effective when they can be evaluated on demand, such that details are synthesized only where they visibly impact the scene. But this lazy evaluation requires a termination test to determine when to stop synthesizing irrelevant details. A geometric termination test requires a bounding volume; the challenge is to predict this bounding volume without synthesizing all the included geometry.

Representations of recursive procedural models include the L-system [18, 17] (a.k.a. the graphtal [20]) and the iterated function system (IFS) [1, 11]. The L-system is more widely used because it is easier to describe and generalizes into a more powerful shape representation. The IFS is more restrictive, which makes it easier to analyze. In many cases we can approximate an L-system with an IFS, then find a bounding volume for the IFS attractor.

Deterministic context-free L-systems (D0L-systems) can be represented by (recurrent) iterated function systems [10, 8] (and vice versa [16].) The recursive self-references found in L-system productions correspond to the iterated maps of an iterated function system.

L-systems often terminate their self-reference through the use of parameterized productions. The resulting object, with its truncated self-similarity, is actually a subset of the corresponding IFS attractor with complete self-similarity. Hence a bounding volume for the IFS attractor suffices as a bounding volume for the smaller L-system shape.

Moreover, the IFS need not contain the L-system model as a strict subset. For example, if an L-system models a tree, then an IFS representation of the tips of the branches plus a point at the base might have an identical bounding volume. A convex bounding volume for the IFS would thus serve to bound the entire tree, even though the IFS attractor may not contain all the details of the tree's branches.

Given an IFS and a desired number of faces n (and their orientations), our algorithm finds the smallest oriented n-faced polyhedron that contains its images under the IFS transformations. Figure 1 illustrates an example of our algorithm bounding the recursive geometry of a fractal tree.

Section 2 reviews previous methods for predicting bounding volumes for procedural models, and Section 3 reviews the iterated function system. Section 4 describes our algorithm in detail, and Section 5 discusses bounding refinement, limitations, and extensions to the RIFS case. Section 6 demonstrates the algorithm and provides results, which indicates that the method is useful for bounding real IFS. Finally, Section 7 concludes and offers ideas for further investigation.

2 Previous Work

While procedural models are much more compact than stored-data models, their construction can be slower due to the time required to execute the geometry-synthesizing procedures. Bounding volumes are often used for procedural models as a way to reduce this processing time. For example, we can use a bounding volume to view-cull offscreen portions of procedural models, or use the size of the bounding volume to terminate the procedural recursion once it reaches pixel size.

Some of the earliest recursive procedural models were midpoint subdivision methods for synthesizing terrain [9]. Analysis of the statistics of the self-affine subdivision led to the construction of the "cheesecake" extent (an extruded triangle) [13] and ellipsoids [4] that contained all possible terrains produced by midpoint subdivision of a patch.

Much of the prior work for predicting the extent of IFS attractors has used bounding spheres. Given a bounding sphere, the transformations of the IFS can generate a bounding volume hierarchy that can be used to efficiently raytrace the attractor [11]. This ray-tracing method depends on the existence of an initial bounding volume, and an ad-hoc algorithm was devised [11] for constructing a loose bounding volume. Given a center (e.g. the mean of the fixed points of each of the the IFS maps), the algorithm determined a sphere radius that would, in the limit, contain its images under the maps of the IFS. More sophisticated methods search for the optimal bounding sphere center that minimizes the radius of the resulting bounding sphere. The mathematical literature shows that this optimal center can be found via the technique of Lagrange multipliers [7]. Graphics papers, apparently developed independently, note that the problem is well suited to the use of a generic nonlinear optimization package [19]. Recent work [15] has improved this optimization process using the invariant measure of the IFS, which allows the anisotropy of the IFS maps to be taken into account. These results appear to be the state of the art in spherical bounds on IFS.

Rather than a bounding sphere, we seek a convex bounding polyhedron. The mathematical literature does contain an iterative method [6] which, if it converges, will find a polyhedral bound. Like our technique, a computer implementation of this iterative method requires the directions, which form the face normals of the bounding volume, to be discretized and fixed beforehand. Although this iterative technique converges quickly for many IFS, there is in general no known limit on the number of iterations required for convergence. By contrast, our algorithm reaches its solution directly, without iteration, since it is based on convex optimization, which has a known maximal run time.

A recent work by Chu and Chen [5] demonstrates a method to construct a tight, axis-aligned bounding box for 2D IFS. Our work, although developed independently, can be seen as a way to generalize this method to more complex bounding volumes and higher dimensions.

3 Iterated Function Systems

An Iterated Function System (IFS) consists of a finite set of functions w_m, which move points around in some space—typically, the functions w_m map $\mathbb{R}^n$ to $\mathbb{R}^n$. For example, the classic Sierpinski gasket's IFS, shown in Figure 2, consists of three maps that contract 2D space by a factor of two towards the points $(0,0)$, $(1,0)$, and $(0.5,1)$.

The Hutchinson operator W maps a set of points to another set of points, and is defined as the union of each of the maps w_m. That is, given a subset of space B,

$$W(B) = \bigcup_{m \in M} w_m(B)$$

Under certain conditions on the w_m,[1] it can be shown that repeated applications of W always converge to a unique attractor A. That is, there exists a set A such that, starting with any bounded nonempty set B, $W(W(\ldots(W(B))) = W^{\infty}(B) = A$. Furthermore, A is invariant under W—that is, $W(A) = A$. Convergent IFS, with a well-defined attractor, are the only IFS that will concern us.

We can now restate a recursive bounding theorem [12, 11], which states that if a shape bounds its image under the operator W, then the shape bounds the attractor, as demonstrated in Figure 2.

[1] A sufficient condition is that each w_m be Lipschitz contractive.

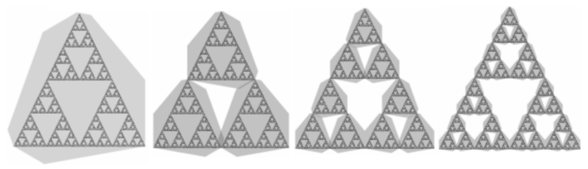

Figure 2. If a bounding volume contains its images under the maps of an IFS, then it contains the attractor of the IFS.

Theorem 1. *Let B be a non-empty compact subset of $\mathbb{R}^n$ and let W be the Hutchinson operator of a convergent IFS on $\mathbb{R}^n$. If $W(B) \subset B$ then $W^\infty(B) \subset B$.*

Proof: A simple proof proceeds by induction on applications of W.

The base case is trivial since $W(B) \subset B$.

For the inductive step, assume for some k, $W^k(B) \subset B$ and let $C = W^k(B)$. The subset property $C \subset B$, is preserved in the image of each of the IFS maps $w_m(C) \subset w_m(B)$, and also in their unions

$$W(C) = \bigcup_{m \in M} w_m(C) \subset \bigcup_{m \in M} w_m(B) = W(B) \subset B$$

Thus $W(C) = W(W^k(B)) = W^{k+1}(B) \subset B$.

By induction, we have $W^\infty(B) \subset B$. $\square$

4 Approach

Our basic approach will be to bound the IFS attractor using a bounding hull built from the intersection of a set of halfspaces. Using the recursive bounding theorem, we will guarantee that the attractor lies within the hull by guaranteeing "containment"—that is, by ensuring that under each map w_m, the map of the hull lies within the original hull.

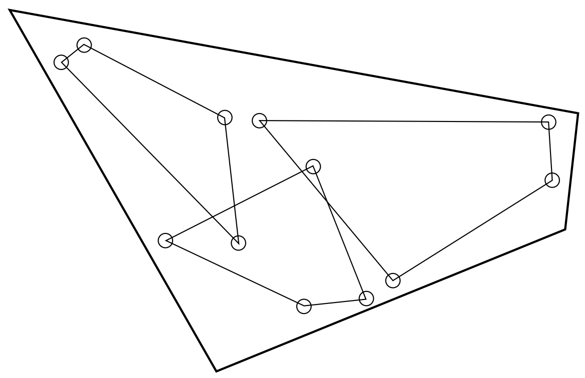

Figure 3. We ensure containment by checking each corner of the hull under each map.

In general, the map of a polyhedral hull is no longer polyhedral; so checking containment can be quite difficult. Luckily, we are normally interested in affine maps in ordinary Euclidean space, so the map of a convex polyhedral hull is still a convex polyhedron. As such, we can guarantee containment by requiring that each corner of the polyhedron (that is, each intersection of the polyhedron's sides), under each map, satisfies all the halfspaces of the hull, as shown in Figure 3.

4.1 Method

Our bounding hull is a convex polyhedron, and hence consists of a set S of halfspaces. Our halfspaces consist of an outward-facing normal $\vec{n}_s$, represented as a row vector, and a scalar displacement d_s. Then we can determine if a point $\vec{x}$, represented as a column vector, lies inside the halfspace by examining the dot product

$$\vec{n}_s \vec{x} \leq d_s$$

In 2D, two halfspaces i and j intersect at a point $\vec{x}_{ij}$ if the point simultaneously satisfies the equations of both halfplanes, so

$$\begin{bmatrix} \vec{n}_i \\ \vec{n}_j \end{bmatrix} \vec{x}_{ij} = \begin{bmatrix} d_i \\ d_j \end{bmatrix}$$

To guarantee containment, and hence apply the recursive bounding theorem, we have to make sure each map of each intersection satisfies each of the halfspaces. That is, given a set of intersections I, maps M, and halfspaces S, we require

$$\forall_{(i,j) \in I, m \in M, s \in S} \quad \vec{n}_s w_m(\vec{x}_{ij}) \leq d_s \tag{1}$$

4.2 Linearity of Constraints

We now need to show how the constraints in equation 1 can be made suitable for use in a linear optimizer. We first note that if we fix the normals and define the matrix

$$\mathbf{N}_{ij} = \begin{bmatrix} \vec{n}_i \\ \vec{n}_i \end{bmatrix}$$

then $\vec{x}_{ij}$ is a linear function of the unknown halfspace displacements d_i and d_j

$$\vec{x}_{ij} = \mathbf{N}_{ij}^{-1} \begin{bmatrix} d_i \\ d_j \end{bmatrix}$$

In 3D, we can similarly define a matrix $\mathbf{N}_{ijk}^{-1}$ to compute the intersection of three halfplanes given their displacements d_{ijk}.

Since we assumed the maps w_m were affine, we can represent each map w_m as a matrix $\mathbf{W}_m$ and a shift vector $\vec{s}_m$, as in

$$w_m(\vec{x}) = \mathbf{W}_m \vec{x} + \vec{s}_m$$

We can now expand out equation 1 and verify that our constraints are now linear in the displacements

$$\forall_{(i,j)\in I,\, m\in M,\, s\in S} \;\; \vec{n}_s \left(\mathbf{W}_m \mathbf{N}_{ij}^{-1} \begin{bmatrix} d_i \\ d_j \end{bmatrix} + \vec{s}_m \right) \leq d_s \qquad (2)$$

This final, linear form of our constraints is suitable for direct use in a constrained linear optimization package; where the displacements d_i are our unknowns and everything else is fixed. Code implementing this set of constraints is listed in Appendix A.

A linear optimization system will also require an objective function. We normally want the "smallest" bounding hull, but exactly what we mean by "smallest" determines our choice of an objective function. Area is a natural choice, but unfortunately the area of the hull is a nonlinear function of the displacements (for example, the area of a $w \times h$ rectangle is wh), so we cannot directly minimize area. Another natural choice is to minimize the largest displacement (for example, we might minimize $\max(w,h)$), which is easy to achieve by the standard technique of adding an additional variable to represent the maximum displacement. Our choice for an objective function is to minimize the sum of the displacements (for a rectangle, $w+h$), which is quite simple and should give results similar to minimizing either area or maximum displacement.

Since many linear optimizers require non-negative values, we can ensure the displacements remain non-negative by expressing the maps w_m in a coordinate system where the displacement origin lies within the hull. For example, we can place the origin at one of the maps' fixed points, since each of the maps' fixed points must lie within the attractor, and the attractor always lies within the hull.

Finally, linear optimization systems can provide a feasibility and optimality guarantee, which means this procedure is guaranteed to find the smallest recursively instantiable bounding hull if any exist. This guarantee is actually not as useful as it may appear, because as shown in the next section, we can often find a substantially smaller bounding hull than the recursively optimal one; and because of corners, a recursively instantiable bounding hull may not even exist.

5 Discussion

This section discusses our bounding hull approach, specifically the process of refining a recursive bounding hull, the inherent limitations of polyhedra bounds under rotation, the fixed orientation and topology of our polyhedra faces, and the extension of the algorithm for recurrent iterated function systems.

5.1 Refinement

Chu and Chen [5] point out that even a recursively-optimal bound, like the one found by our linear optimizer, can still be substantially improved. The reason is that the IFS attractor is defined as the limit, to infinity, of applications of the IFS maps; but a recursively built bound is defined to be big enough to contain its direct images under each IFS map.

This means we can more closely bound the IFS attractor by bounding our very deeply-nested images, which are small, rather than the larger images that are only one level down. In mathematical notation, we should require of our maps B the less stringent property that $W^k(B) \subset B$ for some large k, rather than the unnecessarily limiting recursive bound property $W(B) \subset B$. $W^k(B) \subset B$, like the recursive bound property, is actually still a linear constraint on the bound, so at least in theory we could simply use this new set of constraints in our linear optimizer. But because the number of separate bound images in W^k grows exponentially with k (for an IFS with m maps, W^k maps one point to m^k points), for any reasonable k this refined set of constraints would be too slow to implement directly in the linear optimizer.

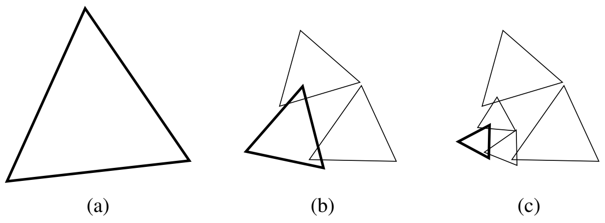

Figure 4. Illustration of the refinement process, while searching for the best left-side bound on this 3-map 2D IFS. (a) is the original bound, which in (b) has been replaced by its maps. Replacing the leftmost bound with its maps results in the new set (c).

Instead, we first use the simpler recursive constraints to achieve a coarse initial bound, then use the iterative refinement approach of Chu and Chen [5] to refine this initial bound, by lazily evaluating the outermost edges of $W^k(B)$. For each halfplane of the refined bound, we search for the outermost image of the original hull, by intelligently traversing the structure of the IFS from the large, initial maps down to the very small deeply nested images–the process is illustrated in Figure 4. Like Hart's original IFS raytracing method [11], the core operation in this refinement process is to find the outermost intersecting bound image, and then replace the bound with its images under the IFS maps.

It may appear that this recursive map-opening process is best implemented recursively, but for IFS where the sub-bounds may overlap a purely recursive implementation may waste a good deal of time computing

```
double planeSearch(IFS, initBound,
            searchPlane, maxDepth)
{
  heap<BoundImage> H(searchPlane);
  H.put(initBound);
  while (O=H.get())
  { /*O is the outermost bound*/
    if (O.depth == maxDepth) break;
    for (each map m of the IFS)
      H.put(O under map m);
  }
  return O.extent(searchPlane);
}
```

Figure 5. Pseudocode for the bound refinement process, using a heap-based lazy evaluation scheme. This search finds the outermost point (relative to an output bound search plane) of an initial bound under repeated maps from the IFS.

very tight bounds that are later discovered to be useless. Instead, a more efficient data structure to organize this outermost-intersecting-bound search is a heap, ordered by distance from the search halfplane, as shown in the psuedocode in Figure 5.

The refined bounding volume is extracted, one half-space at a time, by separate searches for these outermost deeply-nested bounds. The initial bounding volume is represented by its vertices, which makes searching for the outermost point on the bounding volume simple. We find it is much faster if, during the search, instead of actually moving a parent bound's vertices under the maps of the IFS, if we simply move the search plane by the inverse map and leave the vertices stationary.

5.2 Corners

Because unlike a sphere, our polyhedral bounding hull has corners, it may not be possible to bound an IFS by simply increasing the hull's size. For example, consider a single-map 2D IFS that consists of a 30-degree rotation together with a very slight contraction. The attractor for this IFS consists of a single point. However, no 4-sided hull can recursively bound this IFS, because after rotation, the hull's corners will always stick out—see Figure 6. Increasing the size of the box does not help, because rotation and scaling are scale-independent.

Chu and Chen [5] suggest replacing such a problematic IFS with an equivalent one that has more maps but smaller contraction factors. They show that eventually this process will lead to an IFS that can be bounded. Instead, our solution to this corner problem is to add sides—in this case, a 12-sided hull will match itself under rotation, and can then be shrunk exactly to the attractor.

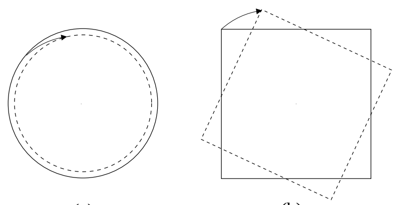

Figure 6. For a one-map IFS, which rotates and slightly scales space, the sphere bound (a) fits nicely; but the corners of a 4-sided box (b) will never fit inside the box. The solution is to use a hull with more sides, like the 12-sided hull in (c).

In 2D, if the angle between two hull corners is α, the corners will never stick out if all the map contraction factors s_m satisfy $s_m < \cos\alpha/2$, as can be seen in Figure 7. Rearranging this equation, we find if the largest contraction factor of any map is s, no corner will protrude if we use at least $h = 2\pi/\alpha > \frac{\pi}{\cos^{-1}s}$ sides.

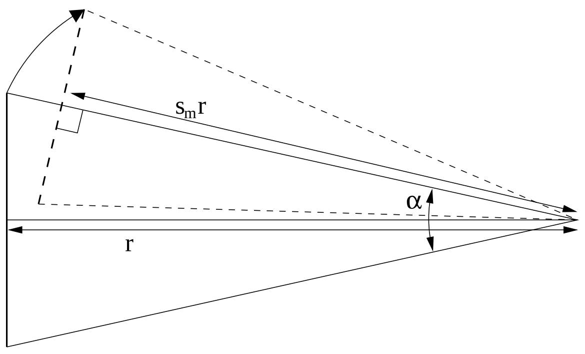

Figure 7. Corners never stick out if $s_m < \cos\alpha/2$.

For contraction factors very close to 1, this means we may require an unaffordably large number of sides to achieve any bound; but as we will show, in practice most IFS only require a very small number of sides. Note that simple translation does not cause this problem, because translation is not scale-independent like rotation. For example, a simple 1D IFS with maps that have a largest scale factor of s and largest shift of d can always be bounded by the interval $[-L, L]$ for any $L \geq |d|/(1-s)$, since then the map of the interval, $[-sL+d, sL+d]$, will lie within the interval because

$$\begin{array}{rcl} |d|/(1-s) & \leq & L \\ -d/(1-s) \leq L & & d/(1-s) \leq L \\ s - d/L \leq 1 & & s + d/L \leq 1 \\ -L \leq -sL + d & & sL + d \leq L. \end{array}$$

5.3 Fixed normals

To implement the algorithm as a linear optimization problem, we first choose fixed normal directions for the faces of our convex hull. A natural choice in 2D, which we have made, is to pick equally spaced directions. In 3D equally spaced normals are more difficult to choose, but we can begin with, for example, the platonic solids.

It is often possible to find a slightly smaller hull via a different choice of face normals, but the gains to be found by optimizing over normal directions are not dramatic, and optimizing over normals is a nonlinear process, so the refinement strategy described above appears to be much more promising.

5.4 Fixed intersections

We must also pick the vertices on the convex hull where faces intersect. In 2D, this is simple–every two adjacent convex hull edges intersect at a hull vertex, and any other edge intersections will lie outside the hull and can be safely ignored. The only way the set of intersections can change is if three edges intersect at a point, in which one of the edges vanishes; in this case we end up checking the same point twice, but this causes no problems.

But in 3D, the set of boundary points is only well defined when exactly three planes meet. In practice, this simply means we are limited to 3D bounding hulls in which every vertex is surrounded by exactly three planes. This property is known as having vertex degree three. For example, a 4-sided tetrahedra, 6-sided cube, or 12-sided dodecahedron all have vertex degree three and are useful bounding volumes. Neither octahedra (vertex degree 4) nor icosahedra (vertex degree 5) are useful bounding volumes, because their vertices are not the simple intersection of three planes, and we have to pick the set of surface vertices beforehand. We can always truncate the vertices of any polyhedron to give a new vertex degree three polyhedron—our implementation repeatedly truncates a tetrahedron or cube until enough sides have been generated.

5.5 Bounding an RIFS attractor

The recurrent iterated function system (RIFS) is an IFS whose maps are controlled by a digraph G whose vertices correspond to the IFS maps. After applying map i, we are only allowed to apply map j if there is an edge in the "control graph" G from node i to node j.

The attractor for a RIFS can be characterized as the union of "attractorlets" A_j [2]. Each attractorlet A_j is the image of one or more attractorlets A_i under the map w_j where an edge from i to j exists in G.

A coarse bound for an RIFS can be found by simply ignoring the control graph G, in effect converting the RIFS into an ordinary IFS. As usual, this requires $|M||I||H|$ constraints. However, a tighter bound can be obtained by independently bounding each of the attractorlets with a separate convex bounding hull. The constraints for such a system would look identical to the constraints seen previously, but since each attractorlet must be bounded separately, this will require $|M||H|$ unknowns (for the $|M|$ attractorlets, each of which is bounded using $|H|$ halfspaces) and $|M|^2|I||H|$ constraints (because for the up to $|M|$ maps of each of the up to $|M|$ attractorlets, each of the $|I|$ intersections must satisfy each of the $|H|$ halfspaces).

6 Results

We implemented this convex optimization bounding method for 2D and 3D IFS. We use a C++ program to generate constraints for the widely available convex linear optimization library lp_solve [3]. In this section, we examine the results and performance of this implementation of the method.

6.1 2D Example

Figure 8 shows the coarse bounding volume computed by our implementation using just 8 sides, and the much tighter bounding volume computed using 30 sides. Both cases visually verify the optimality of the bounding volume, in that any smaller volume with the same number and orientation of sides would not recursively bound the attractor. Both bounding volumes also lie quite close to the actual attractor.

Figure 9 compares out algorithm's output with that of the older sphere radius minimization technique.

6.2 3D Example

Figure 10 shows an important use of 3D bounding volumes computed using this algorithm—to perform raytracing on recursive procedural geometry. We use the raytracing method of Hart and DeFanti [11], which traces recursive geometry by instantiating the bounding volume hierarchy on-the-fly.

Our implementation is reasonably fast, completing an antialiased 500×500 rendering in under a minute on a modern machine. Finding and refining a tight bound takes under 100 milliseconds, so the bounding process is not a significant bottleneck.

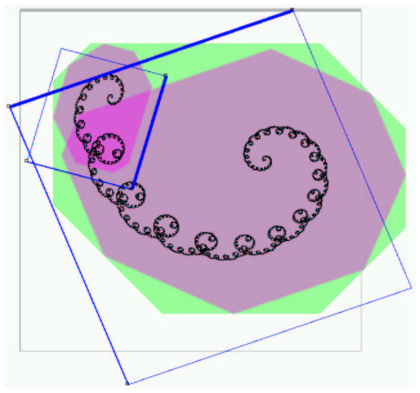
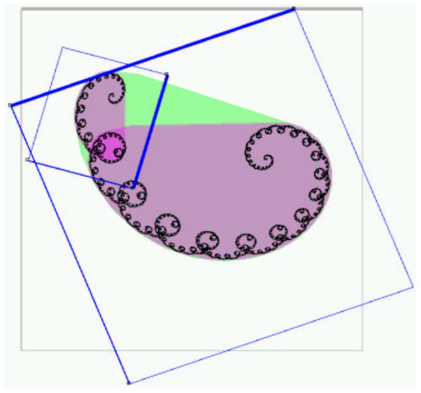

Figure 8. An IFS attractor, with an 8-sided (left) and a 30-sided (right) bounding volume computed by our algorithm.

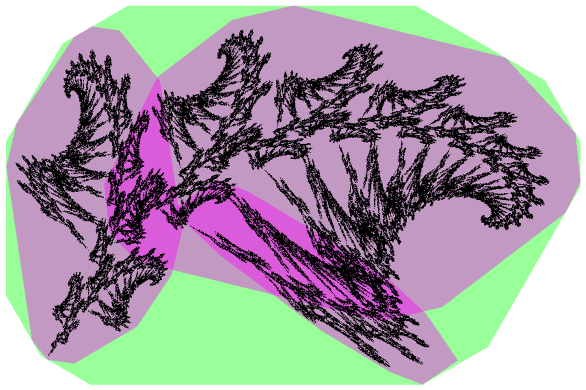
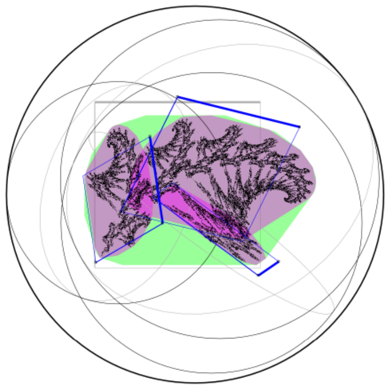

Figure 9. On the left is a 3-map IFS and the 12-sided bounding volume computed by our algorithm. On the right is the much larger sphere bounding volume computed by the older sphere radius minimization technique.

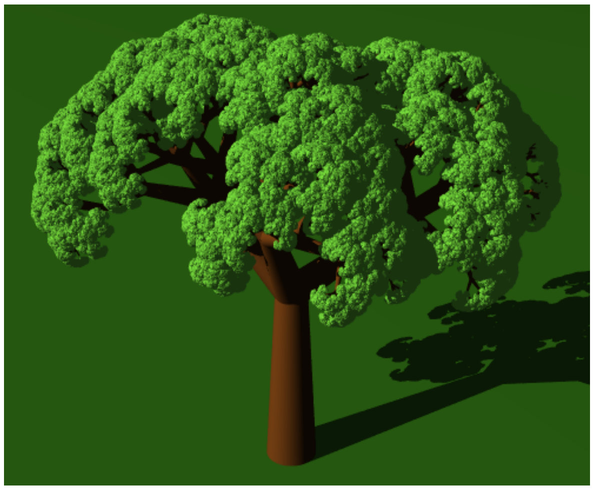
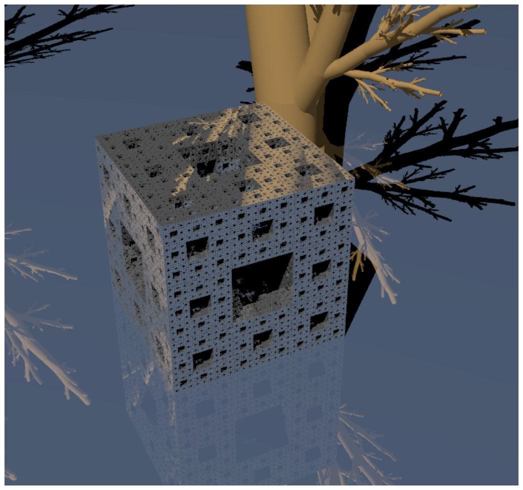

Figure 10. Two raytraced recursive models: on the left, a procedural tree with procedural leaves raytraced using the bounding volume generated by our algorithm. On the right, a reflective Menger's sponge bounded by a tight 6-sided hull.

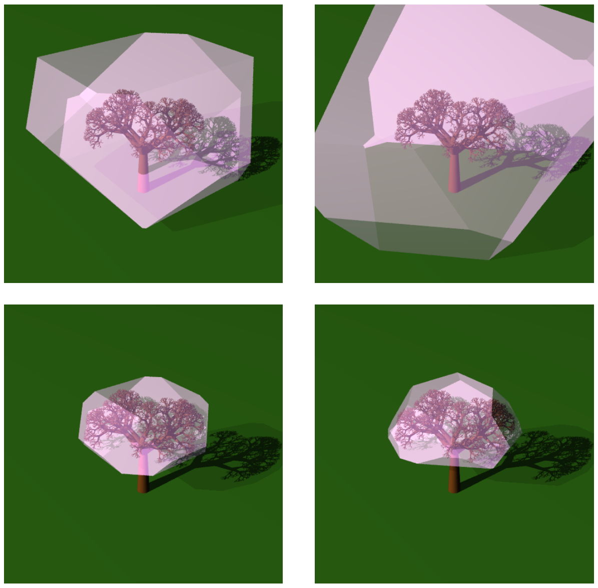

Figure 11. On the left, a truncated cube bound (14 sides); on the right, a twice-truncated tetrahedron bound (20 sides). Top row are the unrefined recursive bounding volumes; bottom row are much smaller refined bounding volumes. All four bounds can obtained and refined in under a second.

6.3 Refinement

Bounding volume refinement, as described in Section 5.1, can substantially reduce the size of recursive bounding volumes. Examples of this for various choices of bounding volume normals are shown in Figure 11.

6.4 Performance

Given an IFS with M maps, and a bounding volume consisting of I intersections between H halfspaces, we will need exactly MIH constraints, of the form given in equation 2, and H unknowns. In 2D, the number of intersections is equal to the number of halfspaces, so this is $O(MH^2)$ constraints.

Although there exist polynomial-worst-case algorithms for solving linear programs, such as the technique of Karmarkar [14]; the package we used, `lp_solve`, is based on the well known exponential-worst-case simplex method. This means our algorithm has a well-defined upper limit, although in theory it may be exponential in the number of bounding volume sides or IFS maps.

The experimental relationship between the number of sides and the run time is shown in Figure 13; the 2D algorithm's total run time appears to be approximately $O(H^{4.6})$, and the run time for the 3D algorithm is similar for a similar number of sides. The 2D IFS for this test are shown in Figure 12, except for "curlyq", which is shown

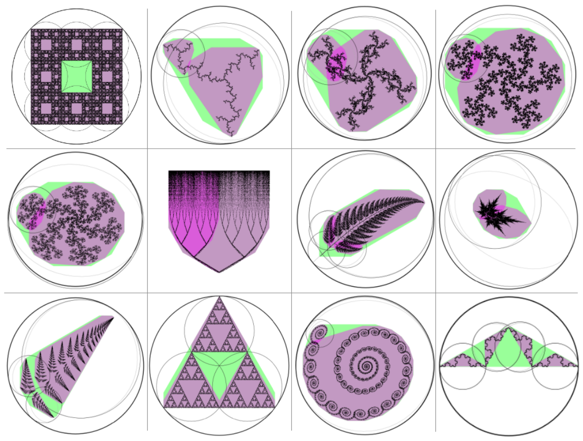

Figure 12. Twelve 2D IFS's, with a loose circle bound found by circle optimization; and the much tighter bound obtained by our method.

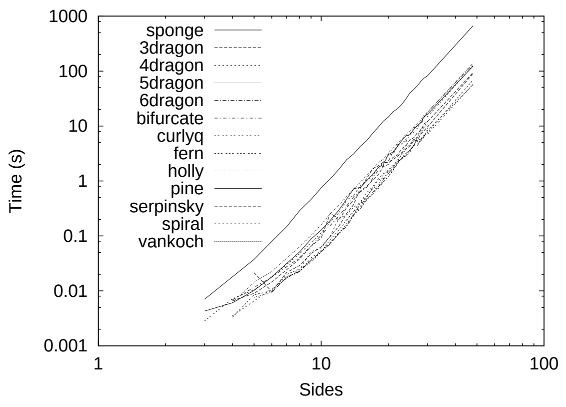

Figure 13. Log-log plot of the time to determine the optimal bound for a variety of sides for a variety of 2D IFS. Runs on a 1.3 GHz AMD Athlon PC running Linux.

in Figure 8. Very large numbers of sides are not computationally feasible at interactive rates; but about ten sides can be computed very quickly. Luckily, as shown below, more sides than this are rarely required.

The experimental relationship between the number of sides and the area of the resulting hull is summarized in Figure 14. Because we distribute the side normals evenly, the area plot jumps up and down as useful normals are found and then passed by. As can be seen in the plot, in practice a fairly small number of sides suffices to bound most IFSs.

Finally, we compared the results of our bounding hull with the bounding spheres of Rice and Hart. The test was a 500x500 rendering of the tree scene shown in Figure 10, without antialiasing (so the total number of rays cast does not depend on the image), with a fixed rendering depth of 10 levels (so the total amount of recursive geometry does not vary), on a 1.6GHz Pentium-M. Table 6.4 compares the raytracing time for the three

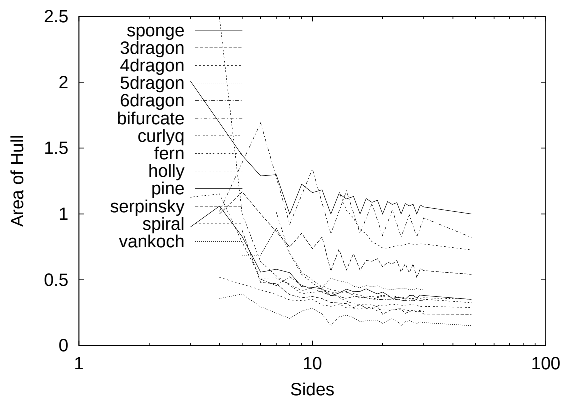

Figure 14. Log-linear plot of the area of the bound found by the algorithm for a variety of sides and a variety of 2D IFS.

Method	*Bound*	*Refine*	*Raytrace*
Truncated Cube	49 ms	13 ms	8.2 s
Rice Sphere	22 ms	-	8.63 s
Hart Sphere	14 us	-	16.0 s

Table 1. Comparing raytracing times for different bounding methods.

methods. Although a ray-sphere intersection is much faster than a ray-truncated cube intersection, the truncated cube is a much tighter fit to the IFS and hence there are many more ray-sphere intersections, making even the best sphere-based methods slightly slower overall. In practice, using an adaptive depth bound would tend to favor even more the tighter bounding volumes found by our method, because a looser sphere bound will have to be more deeply instantiated to reach a given geometric resolution.

7 Conclusions

We have presented a simple algorithm based on convex linear optimization for constructing a recursive convex bounding volume for the attractor of an iterated function system, and then refining that volume. The algorithm is easy to implement, results in tight bounding volumes, and runs at interactive rates for many iterated function systems.

The algorithm can also be used to find bounding volumes for other procedural models, such as the recurrent iterated function system and the L-system. When applied to complex scenes, such as a dense forest of L-system trees, the bounding volumes can be used to prioritize the ordering of the trees and to thus eliminate the unnecessary expense of evaluating an L-system to procedurally produce off-camera or obscured geometry.

7.1 Future Directions

While the equivalence of recurrent iterated function systems and D0L-systems is well understood, there has been little analysis of the geometry of more sophisticated procedural models. General L-systems can contain non-recursive features such as tropism, stochasticism, and context sensitivity that currently elude automatic methods for bounding volume prediction. An IFS can sometimes be constructed from the extremes of these L-system productions, resulting in a worst-case bound on the L-system size. If such an IFS can be found, the bounding volumes described by this paper can be used to bound general L-systems. Although we can currently construct such iterated function systems manually, an automatic technique would significantly extend the state of the art in procedural modeling.

That is, modern procedural models have grown quite complex, with numerous environmental influences that can cause a procedural model to take on many different shapes depending on the context within which it is synthesized. Additional work on more accurately estimating the extent of such geometry would lead to more efficient procedural model rendering, and ultimately to more complex scenes in computer graphics.

References

[1] M. F. Barnsley. *Fractals Everywhere*. Academic Press, New York, 1988.

[2] M. F. Barnsley, J. H. Elton, and D. P. Hardin. Recurrent iterated function systems. *Constructive Approximation*, 5:3–31, 1989.

[3] M. Berkelaar. Mixed integer linear program solver lp_solve. Available from `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`.

[4] C. Bouville. Bounding ellipsoids for ray-fractal intersection. *Computer Graphics*, 19(3):45–51, 1985.

[5] H.-T. Chu and C.-C. Chen. On bounding boxes of iterated function system attractors. *Computers & Graphics*, 27:407–414, 2003.

[6] S. Dubuc and A. Elqortobi. The support function of an attractor. *Numerical Functional Analysis and Optimization*, 14(3&4):323–332, 1993.

[7] S. Dubuc and R. Hamzaoui. On the diameter of the attractor of an ifs. *C.R. Math. Rep. Sci. Canada*, (2,3):85–90, 1994.

[8] D. Ebert, editor. *Modeling and Texturing: A Procedural Approach*. Morgan-Kauffman, 3rd edition, Dec. 2002.

[9] A. Fournier, D. Fussel, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[10] J. C. Hart. The object instancing paradigm for linear fractal modeling. In Proc. of *Graphics Interface*, pages 224–231. Morgan Kaufmann, 1992.

[11] J. C. Hart and T. A. DeFanti. Efficient antialiased rendering of 3-D linear fractals. *Computer Graphics*, 25(3), 1991.

[12] J. Hutchinson. Fractals and self-similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.

[13] J. T. Kajiya. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, 1983. Also appeared in *Computer Graphics 17*, 3 (1983), 91–102.

[14] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–396, 1984.

[15] T. Martyn. Tight bounding ball for affine ifs attractor. *Computers & Graphics*, 27:535–552, 2003.

[16] P. Prusinkiewicz and M. Hammel. Language restricted iterated function systems, Koch constructions and L-systems. In J. C. Hart, editor, *New Directions for Fractal Modeling in Computer Graphics*, pages 4–1 – 4–14. SIGGRAPH '94 Course Notes, July 1994.

[17] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[18] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Developmental models of herbaceous plants for computer imagery purposes. *Computer Graphics*, 22(4):141–150, August 1988.

[19] J. Rice. Spatial bounding of self-affine iterated function system attractor sets. In *Graphics Interface*, pages 107–115, May 1996.

[20] A. R. Smith. Plants, fractals, and formal languages. *Computer Graphics*, 18(3):1–10, July 1984.

A Linear Constraints Code

This piece of C++ code converts the maps of an IFS into a set of linear constraints that require this bounding hull be a recursive bound—that is, that the bound contains its images under each IFS map.

The unknowns for the linear optimizer will be the displacements for each halfspace of our bounding hull. If our normals face outward, then a useful objective function is to minimize the sum of the displacements; this is implemented by adding as the linear optimizer's objective function a long vector consisting of all 1's that will then be dotted with the vector of unknown displacements.

In addition to the number of unknowns and the objective function, the only other item required by the linear optimizer will be the constraints on the unknowns, which are produced by this function. If the linear optimizer requires the unknowns to have only positive values, as discussed in Section 4.2 the IFS maps must be shifted so the coordinate system (and hence displacement) origin will lie within the output hull, then the resulting displacements shifted back after running the linear optimizer.

The inputs to the routine are the maps of an IFS, in the form of a set of affine (translation-free) matrices and shift vectors representing the maps; the normals to the halfspaces of the bounding hull; the topology of the bounding hull in the form of a list of the normals that intersect to form each corner; the number of spatial dimensions; and finally the linear solver to which we will add the constraints. This is essentially a direct implementation of the constraints listed mathematically in Equation 2.

```
void recursiveBoundConstraints(
     int nMaps,// Number of IFS maps
     const Matrix w[],const Vector shift[],
     int nNormals,const Vector normals[],
     int nCorners,const int corners[][],
     int nDimensions, // Spatial dimensions
     LinearSolver &solver
    )
{
  int c,m,s,a;

//Constraints:
  for (c=0;c<nCorners;c++)
  for (m=0;m<nMaps;m++)
  for (s=0;s<nNormals;s++)
  {
  // Faces with indices listed in corners[c]
  //  all intersect to define this corner.
    Matrix N;
    for (a=0;a<nDimensions;a++)
      setRow(N,a, normals[corners[c][a]]);

  // Prepare this linear solver constraint:
    std::vector<double> disp(nNormals,0.0);

    // d_s term, rearranged as -d_s:
    disp[s]-=1;

    // n_s dot W_m N_ij^-1 term:
    Matrix M=w[m]*N.inverse();
    for (a=0;a<nDimensions;a++)
      disp[corners[c][a]]+=
               dot(normals[s],getCol(M,a));

    // n_s dot (... s_m) term, rearranged:
    double lowerBound=
              -dot(normals[s],shift[m]);

    solver.addConstraint(disp,lowerBound);
  }
}
```