# Calculation of Tight Bounding Volumes for Cyclic CSG-Graphs*

**Christoph Traxler** **Michael Gervautz**

**Institute of Computer Graphics**
**Technical University of Vienna**
**Karlsplatz 13/186-2**
**1040 WIEN**
**AUSTRIA**
**email: traxler@cg.tuwien.ac.at gervautz@cg.tuwien.ac.at**

## Abstract

Cyclic CSG graphs are a memory safe representation of objects with a very complex, recursive structure. This class of objects are defined by CSG-PL-Systems, an adaption of the well known Parametric Lindenmayer Systems (PL-Systems) to the CSG concept. They are a powerful tool to model natural phenomena like plants, clouds or fractal terrain but also linear fractals or any objects with a repetitive structure. CSG-PL-Systems are directly translated into CSG graphs, which are a proper object representation for ray tracing. A derivation and geometric interpretation of strings is no longer necessary. Because CSG graphs are as compact as the CSG-PL-Systems the memory usage is low, so that restrictions of the complexity of the scene are avoided. To be efficient as well it is very important to adapt conventional optimization techniques to CSG graphs. For CSG trees a hierarchy of bounding volumes is buildt up by a simple recursive algorithm. A straight forward transition of this algorithm to CSG graphs yields to very huge and thus useless bounding volumes. In this paper we introduce an algorithm which calculates tight bounding volumes for the nodes of cyclic CSG graphs. This method can also be applied to CSG trees with explicit transformation nodes or CSG dags.

**Key Words:** CSG Graphs, Bounding Volumes, Ray Tracing

* This project is supported by the “Fond zur Förderung der wissenschaftlichen Forschung (FWF)”, Austria, (project number: P09818)

# 1 Introduction

Cyclic CSG graphs are a memory safe representation of recursive objects, which are defined by parallel rewriting systems. They are an extention to conventional CSG trees and thus can directly be used for ray tracing. CSG graphs emerge from CSG-PL-Systems, which are an adaption of the well known Parametric Lindenmayer-Systems (briefly PL-Systems). The main difference between PL-Systems, which are fully described in [LIND90], and CSG-PL-Systems are their formal languages. While the derived strings of the first are command sequences for a virtual construction tool called turtle, the derived strings of the last are a subset of the infinite set of all valid CSG expressions. Instead of running a derivation process and building up a large CSG tree out of the resulting string, the rules of the grammar are directly transformed into a cyclic CSG-graph, which is traversed by the rays to visualize the object.

For that purpose, we extended the CSG concept to three new nodes. Selection nodes, briefly S-nodes, join all the rules for one grammar symbol and are associated with a selection function. This function controls the flow by selecting a proper rule and passing the rays to it. The rules themselves are translated into CSG trees and attached as successors to the S-Node. For each occurrence of a grammar symbol in the right hand side of a rule an edge is spanned to the corresponding S-node. This yields to a graph with cycles, wherby S-nodes are the only targets of cyclic edges.

Like with conventional CSG trees, the rays are mapped from the world coordinate frame into the object coordinate frame, because this is much more efficient than mapping the primitive objects. In CSG graphs this is done by special nodes, called Transformation nodes or briefly T-nodes. They can be seen as unary operators and thus have only one successor. The same is true for Calculation nodes, briefly C-nodes, which evaluate a finite set of arithmetic expressions to modify global parameters. Their values effect flow control, i.e. the selection of rules, and transformations. All other nodes, i.e. CSG operators and primitive objects are handled as in CSG trees. The CSG graphs introduced so far are as compact as the describing data set. An explicit representation of the scene (like with CSG trees) is avoided and therefore no restrictions of the complexity of the scene or the approximation accuracy of objects must be considerated. A related method was introduced in [KAJI83] for the ray tracing of fractal terrains and in [HART91] for the ray tracing of objects defined by Iterated Function Systems (IFS). Fig.1.1a shows a CSG-PL-System, that generates a simple sympodial branching structure and Fig.1.1b the corresponding CSG graph. A full description of this approach we gave in [GERV95].

**Initialization of the parameters**

```
{       cnt = 8;                        // order of the branching structure
        trunk = 2,                      // number of trunk segments (cycles of rule with index 2)
        fTR = 0.96;                     // scaling factor for trunk segments
        gamma = -72.0;                  // divergence angle
        alpha1= -60.0;                  // branching angles
        alpha2 = 20.0;
        fBR1 = 0.73;                    // scaling factors for left and right branches
        fBR2 = 0.67;
        noleaves = 3;           // if cnt is less than noleaves the limbs bear leaves
        segments = 2;                   // number of segments for a limb with leaves
        fxSG = 1/7;                     // scaling factors for segment cylinders
        fySG = 1/7;
        dSG = 1/segments;               // height of a segment cylinder
        fLV = 0.6;                      // scaling factor for leaves
} TR                                    // the start node
```

**The rules for TR and SG**

TR if(trunk > 0, 2, if(cnt > 0, 3, 1)) // module for the tree

1: TR → SG // terminating rule

2: TR → SG ∪ {trunk=trunk-1} move(0,0,1) uscale(fTR) TR // rule for the generation of the trunk

// rule for the generation of the branching structure

3: TR → SG ∪ move(0,0,1) rotz(gamma)({cnt=cnt-1} rotx(alpha1) uscale(fBR1) TR ∪
{cnt=cnt-1} rotx(alpha2) uscale(fBR2) TR )

SG if(cnt < noleave, if(segments > 0, 3, 2), 1) // module for segments

1: SG → scale(fxSG,fySG,1) cylinder // this rule is selected for limbs without leaves

2: SG → segment // terminating rule for limbs with leaves

3: SG → segment ∪ {segments=segments-1} move(0,0,dSG) rotz(-90) SG // generating rule for limbs with leaves

// CSG expression for segments with leaves

segment = scale(fxSG,fySG,dSG) cylinder ∪ move(0,0,dSG) (uscale( fLV) leaf ∪ flipxz uscale(fLV) leaf)

**Fig. 1.1a :** A PCSG-system for a simple sympodial branching structure. (Taken from [GERV95])

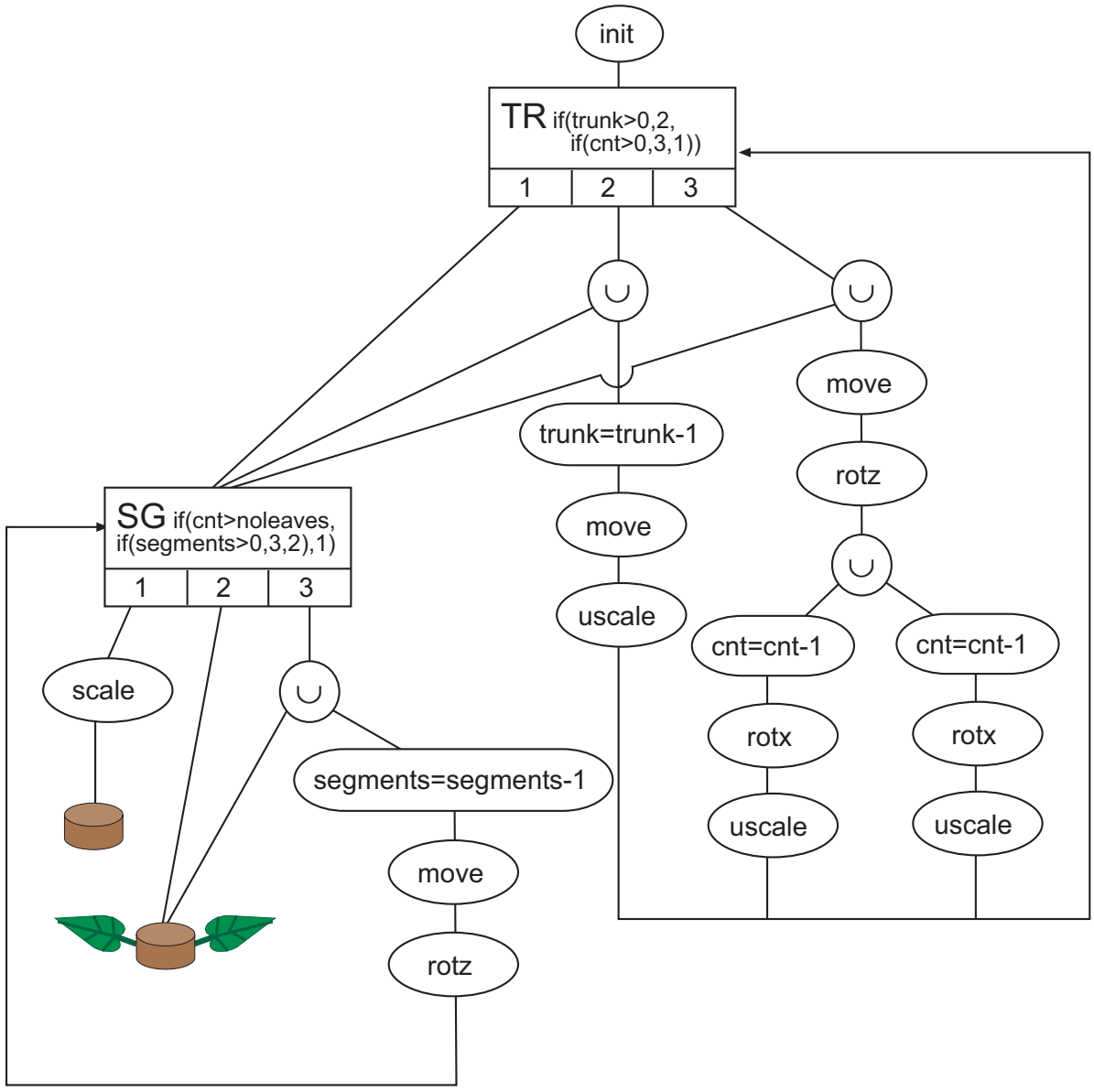



**Fig. 1.1b :** The CSG graph that generates the sympodial branching structure. (Taken from [GERV95])

To make the method not only memory safe but also efficient, it is very important and essential to optimize ray tracing. Bounding volumes are used for conventional CSG trees to enclose parts of the scene and to test the rays against them. They are calculated for each node to obtain a bounding volume hierarchy. The traversal of a ray is terminated as soon as it misses a bounding volume. While the calculation of bounding volumes in a preprocessing step is a quite simple task for CSG trees, it is not trivial for cyclic CSG graphs. The purpose of this paper is to describe a method for the calculation of tight bounding volumes for CSG graphs. This method can also be applied to CSG trees with internal T-nodes or CSG dags. We first explain why the conventional algorithm fails to produce tight bounding volumes for CSG graphs. After that we introduce and analyze our approach.

## 2 Problems with the calculation of bounding volumes for CSG graphs

The most common type of bounding volumes are axis aligned boxes. The effort to combine two boxes and to intersect a box with a ray is quite low. In the remainder of this paper we focus on that kind of bounding volumes without loss of generality. The CSG tree is recursively traversed from left to right, to obtain a bounding box for each node. A leaf node consists of a primitive object and an affine transformation, whereas an internal node is associated with one of the three Boolean operators $\cup$, $\cap$, and $\setminus$. When the recursion terminates at a leaf node, a bounding box is calculated that encloses the transformed primitive object. An internal node combines the bounding boxes of its left and right successor according to its Boolean operator. In this simple way a hierarchy of tight bounding boxes emerges. Nevertheless the bounding boxes for CSG graphs obtained in this fashion are very huge, so that their aim completely gets lost.

If a CSG graph is unfolded into a CSG tree it is obvious that almost each node of the graph corresponds to more than one node of the tree. That means that these nodes represent different parts of the scene. Moreover a particular part of the scene is iteratively mapped into different coordinate frames by T-nodes and iteratively combined with other parts by CSG-operator nodes. From an algorithmical point of view, each node of the CSG graph gets a couple of different bounding boxes during the recursive traversal.

A bounding box of a node representing different parts of the scene must enclose all of them. There are at least two possible solutions. The first way is to use an array to store all the bounding boxes for the different parts of the scene. The second way is to join all of them to a single bounding box. The first solution needs a vast amount of memory, comparable to the unfolding of the CSG graph into a huge CSG tree. Thus the advantage of a memory safe data structure would get lost, so that only the second solution remains. We call the union of all bounding boxes for a node its hyper bounding box. With this notation we can now describe how the algorithm for CSG trees is adapted to the internal nodes of a CSG graph (there are no changes for primitive objects):

- Wait for the bounding box(es) of the successor(s). It is important to note that the hyper bounding boxes are not passed back to the predecessor but those which is joined with it. An operator node combines the bounding boxes of its successors according to its Boolean operator and calculates a new one that encloses the result. A T-node has to map the bounding box of its successor and calculates a new one, that encloses the transformed box. This is necessary because the transformation usually destroys the geometric properties of an axis aligned box. C- and S-nodes simply take over the bounding box of the sucessor without any changes. (For S-nodes this is the currently selected successor).

- Join this bounding volumes with the hyper-bounding volume to generate a new one.

As we will see later the hyper bounding boxes are not the severe problem of this algorithm. The major problem, which demands a special algorithm for CSG graphs evolves out of both, the simple geometry of the bounding boxes and the iterative transformations. The rotation of an axis aligned

box by a T-node violates its geometric properties, because its edges are no longer parallel to the world coordinate system. As briefly stated above the T-node has to calculate a new axis aligned bounding box, that encloses the rotated one. Since transformations are applied iteratively the T-nodes have to calculate new boxes with the required properties several times. This yields to a sequence of rapidly growing bounding boxes as shown in Fig.2.1. The resulting bounding box is huge and completely useless.

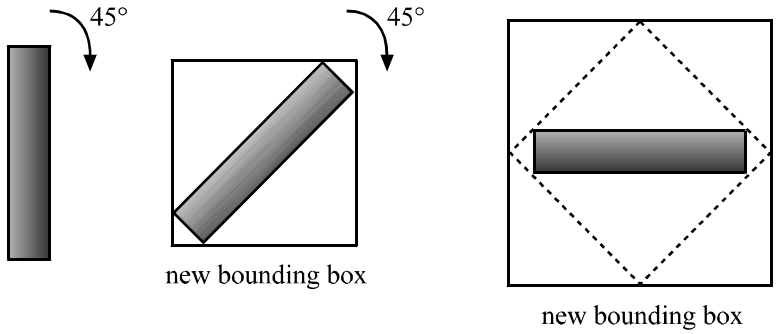



**Fig. 2.1 :** Growing bounding boxes during 2 successive rotations about 45°. This angle has the highest growth rate for axis aligned bounding boxes. (Taken from [GERV95])

The use of other bounding volumes than axis aligned boxes can scarcely temper the growth effect. Even parallel planes with fixed orientations as introduced in [KAY86], which allow very tight enclosures, would grow too much during successive rotations that violate the orientation constraints. Bounding spheres are invariant under rotation, but a similar growing effect can be achieved by successive nonuniform scalings. Beside that spheres are unsuitable to join them to a hyper bounding sphere, because the union of two spheres can be quite large.

None of the mentioned problems occure if all the transformations used in a CSG graph are contractive. In this case they form an IFS and the combination of the bounding boxes in an arbitrary recursion depth perfectly fits into the bounding box of a lower recursion depth. This becomes clear when looking at the Sierpinski-tetrahedron, which consists of smaller copies of itself. Fig.2.2 shows the CSG graph that generates this classical fractal. Imagine a recursion depth where each of the four T-nodes gets the bounding box of an approximated Sierpinski-tetrahedron. Each T-node creates a box of halve the size and moves it to the proper subpart of the tetrahedron. The final combination of these boxes by the $\cup$-nodes results in a box of exactly the same size as that, which was passed to the T-nodes before.

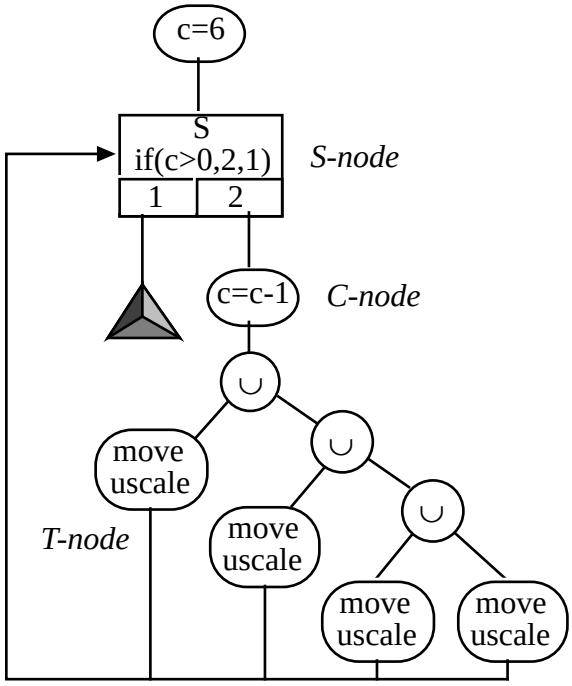



**Fig. 2.2:** A CSG graph representing the Sierpinski tetrahedron (taken from [GERV95]).

Modeling with PL-Systems is a non trivial task. It usually takes a lot of time to specify a PL-System that generates the desired object. A restriction to contractive transformations makes this task unacceptable complex. The automatically translation of an arbitrary PL-System into an equivalent one with contractive transformations or into an IFS, is still an unsolved problem. It is discussed in chapter 8.2 of [LIND90].

# 3 An algorithm for tight bounding volumes for CSG graphs

The key idea behind the new algorithm is to avoid the simple passing of bounding boxes from a successor to a predecessor node. Especially T-nodes have to calculate their bounding boxes directly out of the geometry of the primitive objects, whereby succeeding transformations have to be taken into account. Since CSG graphs have cyclic edges a T-node can be instanced several times. An instance of a T-node maps an object from a coordinate frame $C_k$ into another coordinate frame $C_{k+1}$. $C_0$ is the object coordinate system of a certain primitive object. Thus $C_1$ is the local coordinate frame defined by the instance of that T-node, which transforms the object first. $C_n$ indicates the world coordinate system if *n* transformations are necessary to map the object into it.

A CSG graph defines a plenty of cyclic paths from the root to a primitive object. Along these paths there are many instances of one or more T-nodes. Let us first ignore the binary CSG-operators and extract such a path as transformation chain $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow ... \rightarrow C_{n-1} \rightarrow C_n$. We now have to calculate a bounding box for each coordinate frame $C_i$ directly out of the geometry of the primitive object defined in $C_0$. The geometry of an object is defined by a set of points that approximate its convex hull sufficiently. An axis aligned bounding box is simply derived by finding the minimal and maximal coordinates of the point set. Let $T_i$ denote the transformation of that T-node instance, which performs the mapping from $C_{i-1}$ to $C_i$. To calculate a bounding volume for $C_k$ we just have to map the point set from $C_0$ into $C_k$ by applying the transformations $T_k \times T_{k-1} \times ... \times T_1$. This bounding box is joined with the corresponding T-node's hyper bounding box, because there is a 1:n relation between T-nodes and coordinate frames.

The algorithm works with a stack, which is flexible enough to contain both, transformations and bounding boxes. Now a T-node has to perform the following steps:

1) Push the transformation onto the stack

2) Recursive call for the successor

3) Pop the bounding box from the stack

4) Join it with the hyper bounding box and pass it back to the predecessor

When the recursion terminates at a primitive object, all the bounding boxes for the different coordinate frames defined by the transformations in the stack are calculated as described above and pushed onto the stack. To avoid a great number of matrix multiplications we use a class hierarchy for affine transformations, so that they can be handled like functions. To pass a point P through a transformation chain $T_3 \times T_2 \times .T_1$, we simply evaluate $T_3\,(T_2(T_1(P)))$. Step 4 is necessary to deliver the actual bounding box to predecessor nodes, which are not T-nodes. All the instances of C-, S- or CSG-operator nodes on the path back to the next instance of a T-node belong to the same coordinate frame.

What remains to complete the algorithm is the consideration of the binary CSG operator nodes. In conventional CSG trees these nodes combine the bounding boxes delivered from their successors according to their Boolean operator. The only difference in our algorithm is, that CSG-operator nodes have to combine two stacks of bounding boxes. Instead of a transformation chain we now have to deal with a binary tree of transformations. That means that the bounding boxes for all coordinate frames on the path to a branching point are the combined bounding boxes of the stacks from the left and right path. Finally CSG-operator nodes calculate the combination of the boxes delivered by their successors to get their own actual bounding box, and join it with their hyper bounding box.

## 4 Results

The algorithm described in the last chapter calculates tight bounding boxes for each coordinate frame. In CSG trees with internal T-nodes or in CSG dags they enclose perfectly the corresponding parts of the scene. There is an 1:1 relation between T-nodes and coordinate frames, so that exactly one bounding box is relevant for each T-node. For CSG graphs however this is not true. Here we have an 1:n relation, which demands that *n* bounding boxes have to be joined to a single hyper bounding box for the most T-nodes. Hyper-bounding boxes only fit to zero level instances of the nodes, for instances of deeper recursion depth they are too large. Fortunately this problem is diminished by an essential aspect of CSG graphs, - the iterative transformations.

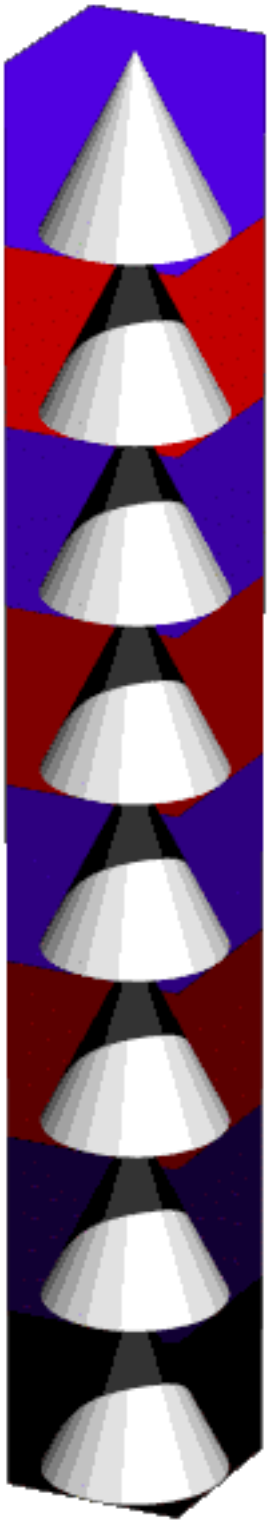
The rays are transformed from one coordinate frame into another. Thus the geometric relation of a ray to one and the same hyper bounding box changes with every mapping. Since this relation is symmetric, we can also say, that constant rays are tested against iterativeley transformed boxes. If a ray hits the same hyper bounding box in two succeeding coordinate frames $C_i$ and $C_{i+1}$, than we can conceive that as hits of two different boxes $B_i$ and $B_{i+1}$, defined relatively to $C_i$ and $C_{i+1}$. This means that the ray hits the intersection of both boxes $B_i \cap B_{i+1}$. Thus the hit of a hyper bounding box at a certain recursion level *k* can be seen as the hit of the intersection of all boxes encountered on the path from the zero level to that recursion level, i.e. $B_0 \cap B_1 \cap ... \cap B_{k-1} \cap B_k$. This intersection gets smaller with increasing recursion depth, so that a ray has a new chance to miss one and the same hyper bounding box after each transformation into a new coordinate frame. Therefore hyper bounding boxes are usefull for deeper levels of recursion as well.

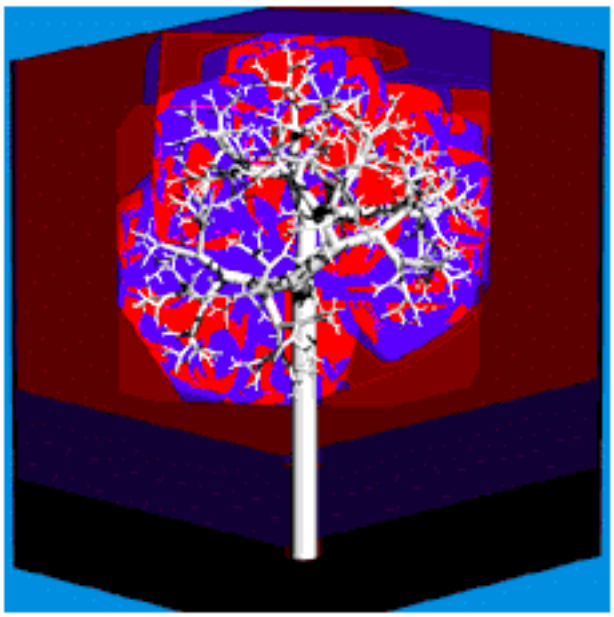
The color plate shows the bounding box hierarchy of three different objects. When the current ray misses a bounding box the background is colored according to the recursion depth. The brighter the color the deeper is the level of recursion at which the corresponding ray misses the bounding box. The difference between red and blue has just the purpose to enhance the contrast. The pictures show that there are rays which miss a hyper bounding box at deeper recursion levels.

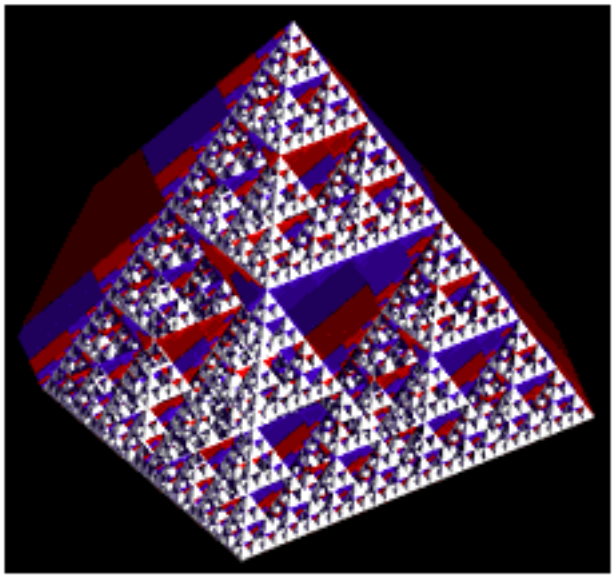
## 4 Conclusion and future work

The calculation of tight bounding boxes for CSG-graphs makes this data structure not only memory safe but also efficient. Thus very complex scenes consisting of recursive objects can be rendered with ray tracing in a reasonable time. The class of recursive objects includes natural phenomena like plants or fractal terrain, as well as the whole set of linear and stochastic fractals. The algorithm generates bounding boxes which can be iteratively used to terminate the recursive traversal of rays, which reduces the computation time drastically. The necessary hyper bounding boxes are not a severe problem. When a ray is mapped into another coordinate frame by a T-node, it has a new chance to miss the same hyper bounding box it hit in the last cycle. Beside that the algorithm can easily be adapted to calculate other kinds of bounding volumes out of the transformed point sets.

Currently we investigate the use of a 3d-grid as additional optimization technique. Non empty voxels represent a state of a CSG graph, which is defined by a transformation matrix, a set of parameter values and a reference to a node of the graph. The foremost, non empty voxel is determined by using 3DDDA as described in [FUJI86]. Now a ray can entry the CSG graph to an advanced sate, which prevents it to run through all the cycles that are necessary to reach this state. In the best case the ray is directly passed to a primitive object, i.e. a terminal node of the graph. In the most cases the entry point will be an intermediate state, so that further cycles are necessary. The grid is filled during the preprocessing step using the bounding box stack.







**Color plate:** the cone tower (left) is a very simple repetitive structure. It shows the increasing level of recursion and that in each level rays are discarded because they miss the hyper bounding box in that coordinate frame. The same can be observed for the simple sympodial tree structure (top right) and the Sierpinski-tetrahedron (bottom right).

## References


[FUJI86] Fujimoto A, Tanaka T, Iwata K
ARTS: Accelerated ray-tracing system.
IEEE Computer Graphics & Applications, Vol. 6(4), pp. 16-26, 1986

[GERV95] Gervautz M., Traxler C.
Representation and Realistic Rendering of Natural Scenes with Cyclic CSG graphs
accepted for publication in Visual Computer, 1995

[HART91] Hart J.C, DeFanti T.A
Efficient antialiased rendering of 3d linear fractals.
ACM Computer Graphics SIGGRAPH Proc., Vol. 25(4), pp. 91-100, 1991

[KAJI83] Kajiya J.T
New techniques for ray tracing procedurally defined objects.
ACM Transaction on Graphics, Vol. 2(3), pp. 161-181, 1983

[KAY86] Kay T.L, Kajiya J.T
Ray tracing complex scenes.
ACM Computer Graphics SIGGRAPH Proc., Vol. 20(4), pp. 269-278, 1986

[LIND90] Prusinkiewicz P, Lindenmayer A
The algorithmic beauty of plants.
Springer Verlag, New York, 1990